Optimization of OpenCL applications on FPGA

Author:
Albert Navarro Torrentó

Master in Investigation and Innovation

2017-2018

Director:
Xavier Martorell

Co-director:
Daniel Jiménez

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech
Barcelona Supercomputing Center (BSC-CNS)

This work has been supported by the Ministerio de Economia y Competitividad under contract Computacion de Altas Prestaciones VII (TIN2015-65316-P), and the Departament d'Innovacio, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programacio i Entorns d'Execucio Paral.lels (2014-SGR-1051), and the Intel Exascale R&D Lab collaboration between Intel and the Barcelona Supercomputing Center.

Abstract

Since Moore's Law is over, specialized accelerators have becoming more and more trending over the years. FPGA is one of this accelerators and their "reconfigurable hardware" capabilities make it really promising. FPGA are programmed with HDL languages which is hard and time-consuming so many high-level alternatives (such HLS, OpenCL, SystemC, ...) have emerged to provide a better performance/development time ratio. This document presents a performance and energy comparison between several algorithms on FPGA using OpenCL and we compared with CPU versions using OpenMP and OmpSs. We concluded that FPGA provide better power efficiency than CPU and it can even have better performance in some cases. Our NBody implementation achieve a speedup of 3x over the CPU and 17x in energy consumption.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Moore's law is ending, which means that performance, although essential, will not longer scale that much by integrating more transistors on a single chip. Also power has become a important factor since the rise of smartphones and portable devices. The way to achieve this objectives, is to create networks of many-cores with multiple accelerators. This is clear on the Top500 where half of the Top10 use accelerators to boost the performance.

GPU is the most prolific and popular accelerator nowadays. It provides great performance for very parallel programs and device programming is easy using languages like OpenCL and CUDA. The usage of runtimes like OmpSs or the unified memory provided by CUDA 8.0 and OpenCL 2.0 makes the transfers to the device transparent, so they are even easier to program

FPGAs are the new accelerators that are becoming trending. They provide a great performance/watt ratio but they need to be programmed in a HDL (Hardware Description Language) which is hard and slow. That's why vendors are developing compilers to support high-level languages, such as OpenCL or HLS, to speedup the development. With OpenCL, even communication with the device is abstracted so the host code is portable. Since each device (CPU, GPU and FPGA) have a different way to exploit their performance, OpenCL programs are not always performance portable.

In this work we present an evaluation of the possible optimizations and porting of several applications to FPGA using OpenCL. Results of this evaluation are several accelerated applications and a set of lesson learned.

1.1 Motivation

FPGA are becoming more accessible with the introduction of OpenCL and HDL. This opens a gateway to be used massively in fields which require accelerators such as HPC. This fields needs to maximize the performance that the device can offer, so studying which type of programs and which practices perform well on the FPGA is necessary to exploit all the device features.

1.2 Objectives

The main objective of this master thesis is to evaluate and analyze OpenCL as a mechanism to exploit the resources of the FPGA to obtain performance of the applications. In order to do this evaluation, this master thesis provide a set of optimized OpenCL programs that exploits the performance of an Altera Arria 10 FPGA on a shared memory architecture.

Chapter 2

Background

2.1 OmpSs

The OmpSs programming model [3], developed at Barcelona Supercomputing Center (BSC), integrate different features from the StarSs programming model family into the OpenMP standard. It is composed of the Mercurium source-to-source compiler and the Nanos runtime.

Instead of using a fork-join model like OpenMP, it uses a thread-pool model. It defines the *task* directive to set the units of work that will be executed by the thread-pool. The programmer can indicate a list of input/output dependencies that allow to determinate task scheduling.

One of its novel features is the *target* directive. This directive allows to define in which device the task needs to be executed, allowing the programmer to abstract from all the device communication. We relay in this directive and in the Nanos OpenCL backend (OmpSs@OpenCL) to avoid the need to add verbose OpenCL code in the host code and speed up the development.

Nanos is integrated with Extrae; a library developed at Barcelona Supercomputing Center (BSC), that allows to extract information from the parallel execution, so you can detect problems with your parallel design.

```
void Cholesky(int NT, float *A[NT][NT] ) {
   for (int k=0; k<NT; k++) {
      #pragma omp task inout ([TS][TS](A[k][k]))
      spotrf (A[k][k], TS) ;
      for (int i=k+1; i<NT; i++) {
        #pragma omp task in([TS][TS](A[k][k])) inout ([TS][TS](A[k][i]))
        strsm (A[k][k], A[k][i], TS);
      }
      for (int i=k+1; i<NT; i++) {
        for (j=k+1; j<i; j++) {
          #pragma omp task in([TS][TS](A[k][i]), [TS][TS](A[k][j])) \
                           inout ([TS][TS](*A[j][i]))
          sgemm( A[k][i], A[k][j], A[j][i], TS);
        }
        #pragma omp task in ([TS][TS](A[k][i])) inout([TS][TS](A[i][i]))
        ssyrk (A[k][i], A[i][i], TS);
      }
   }
}
```

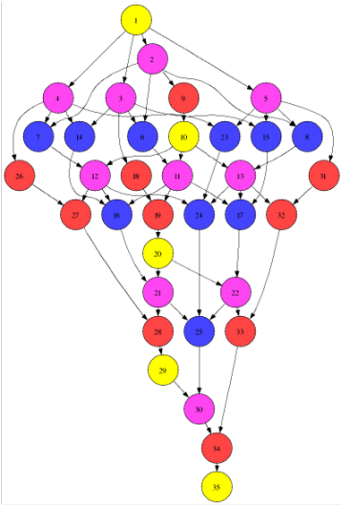


Figure 2.1: OmpSs programming model

2.2 FPGA

Field Programmable Gate Arrays (FPGAs) are semiconductor devices composed by Logic blocks interconnected by a network. This blocks contains a small memory called Look-Up Table (LUT), a register and a multiplexor. The LUT is the key component. This memory can be configured to get a specific output for a specific input, recreating any function that we need. The register stores values and the multiplexor helps to choose the output of the block: the LUT or the register.

The Logic block, is the FPGA most basic unit. Modern versions of this devices usually contain additional items as small memory blocks (BRAMs) and DSPs. DSPs are blocks designed to perform complex arithmetic operations that require many logic blocks.

This device need a connection with a memory where you get the data. This one can be shared with the CPU or dedicated for the FPGA and filled by the CPU with a PCI or QPI bus.

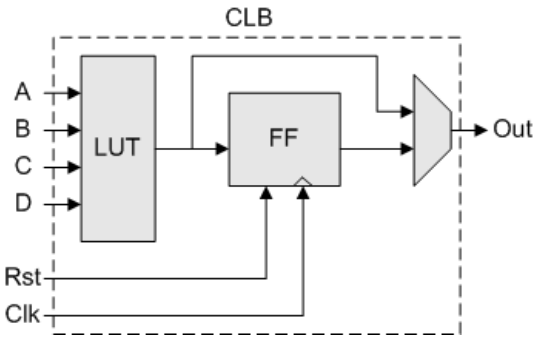


Figure 2.2: LUT

2.2.1 Intel/Altera Arria 10

Our FPGA is an Intel/Altera Arria 10 GX1150, sharing the host memory with a QPI bus providing a bandwidth about 20GB/s. This FPGA has the follow specifications:

LUTS	427,200
Registers	1,708,800
BRAMs	2,713
DSPs	1,518

Figure 2.3: Board resources

In this device some parts of the communication are not implemented in ASIC. Instead of that, there is a base bitstream that contains the communication modules. This implies that part of the board is not available for us.

LUTS	193,220 (49%)
Registers	295,440 (17%)
BRAMs	637 (23%)
DSPs	183 (12%)

Figure 2.4: Board interface area usage

2.3 OpenCL

Modern architectures designs exploits parallelism as the main way to increase performance. CPUs add cores, GPUs add compute units, FPGAs exploit pipeline

parallelism... Exploiting correctly an heterogeneous architecture with this devices is not trivial. CPU parallel models usually assume a shared memory space and out-of-order operations. GPUs address complex memory hierarchies and have very specific vendor extensions. The approaches are really different and this limitations make difficult to software developers access to the device computer power.

OpenCL [9] is a general propose standard for parallel programing targeting CPUs, GPUs and other devices, providing a portable language and a common interface to program heterogeneous platforms.

OpenCL provides a host API to manage devices tasks and a C99-like language to define the kernels. OpenCL kernels are defined as C functions with memory buffers as inputs and outputs. Inside the kernels, OpenCL defines parallelism in 2 ways: work-groups and work-items. Work-items are similar to CPU threads. They have 3 IDs (x,y,z) associated so the programmer can write code that distribute the work between the work-items. Also, work-items are grouped in work-groups with additional IDs to distribute the work among the groups. Work-groups are independent between them, there is no synchronization method, but work-items can be synchronized inside the same work-group.

OpenCL devices contain multiple compute units. Work-groups are assigned to a free compute unit. The programmer must define enough work-groups to exploit all the available compute units, and enough work-items to exploit the compute unit resources.

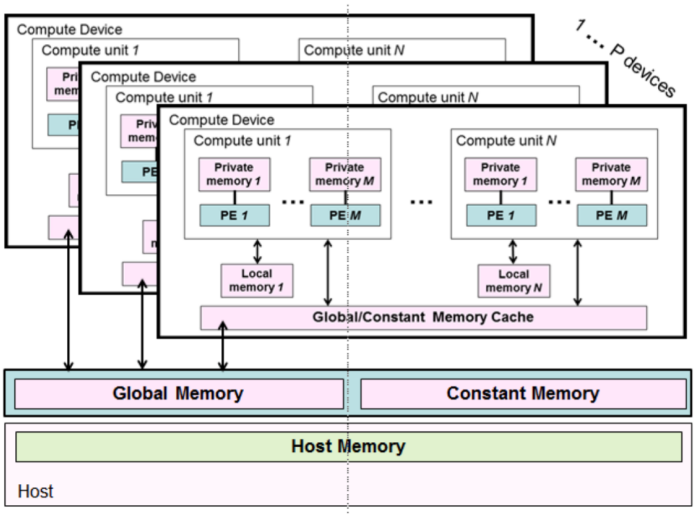


Figure 2.5: OpenCL memory model

2.4 Intel OpenCL on FPGA

Intel provides an environment to generate bitstreams from OpenCL. A source to source compiler generates Verilog from OpenCL code and sends it to the Intel standard toolchain to generate the bitstream (Figure 2.6).

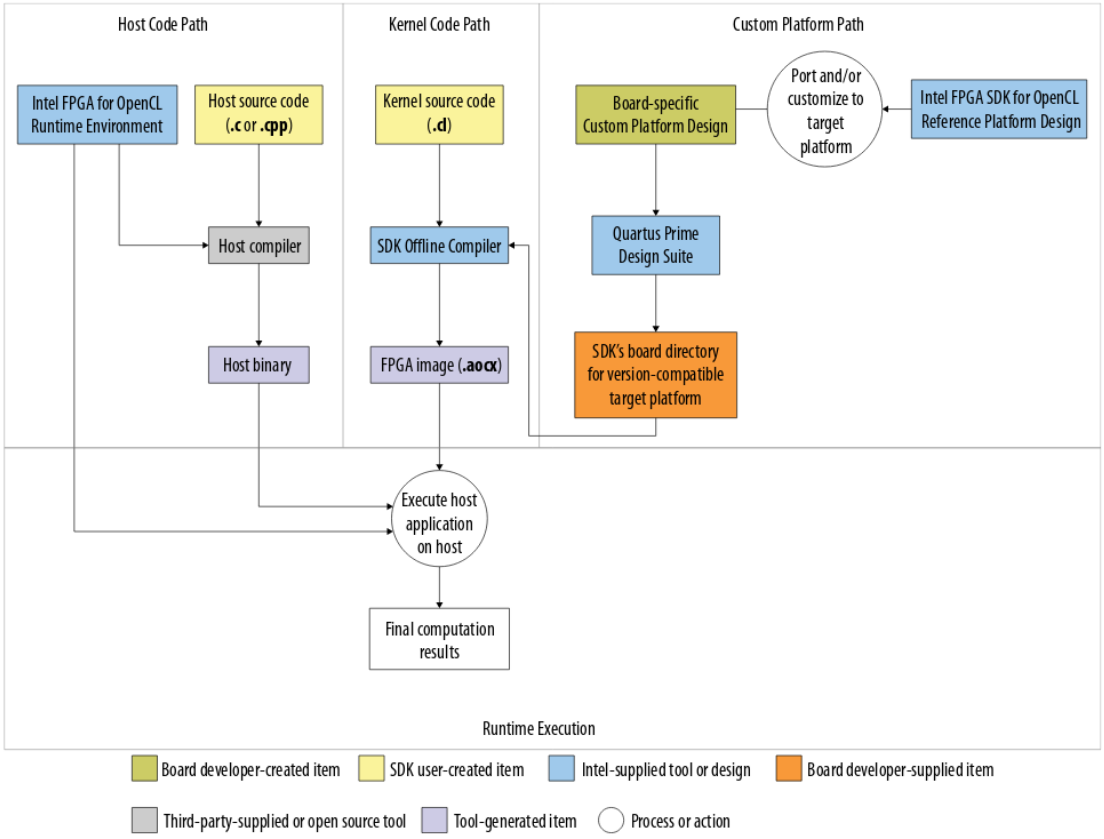


Figure 2.6: OpenCL FPGA compilation [1]

OpenCL language is more intended for a GPU, but the language behavior can be ported quite well to an FPGA. The basic idea is that, for each kernel, the OpenCL compiler will generate a pipeline from the OpenCL code. This is intuitive until we add the language most complex structures: conditionals, loops and also specific features of OpenCL: work-items and work-groups.

The following descriptions are valid for Intel OpenCL compiler. There is no standard way on how are this language maps on the FPGA, but other vendors like Xilinx compile OpenCL mostly in the same way.

NDRange and task

A task is a kernel that has a required work-group size of 1. It acts as a single-thread kernel. We will refer to kernels with work-groups greater than 1 as

[1] Figure from Intel FPGA SDK for OpenCL https://goo.gl/Ybz9LV. Page 7

NDRange kernels or threaded kernels. The compiler makes distinction between them when you have loops. We will talk about that in a future section.

Conditionals

Conditionals can't be like a CPU or GPU. We don't have a program counter to change, we only have a long pipeline. The compiler approach is a more hardware like. The compiler will take both paths of the conditional and will select the result at the end (A hardware multiplexor) like in Figure 2.7. This means that conditionals have no penalization for "jump miss prediction" like in CPUs or "serialization" on GPUs.

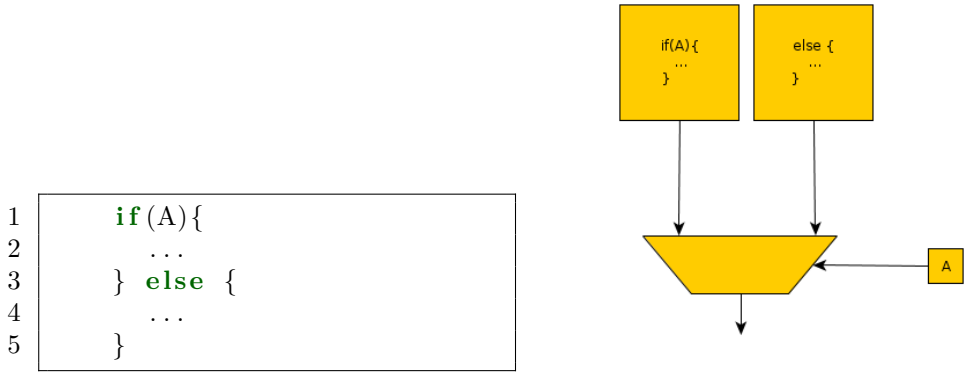


Figure 2.7: Conditional OpenCL to Verilog

Loops

Loops can be implemented with some kind of FSM. The structure must contain some control logic for dependencies and also needs a feedback logic to return the loop carried dependencies. Notice that we will always have carried dependencies (at least the loop counter like, in Figure 2.8).

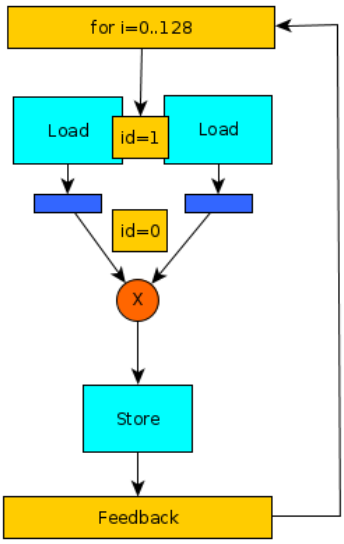


Figure 2.8: FPGA loop behavior

To maximize the hardware occupancy, the compiler always tries to pipeline the loops in single-thread kernels (Figure 2.9). This allows to launch an iteration before the previous one has finished. The rate in which a new iteration is launched (Initialization Interval(II)) depends on the carried dependencies. If the iterations depended between them, you need to wait until the dependency is resolved, Also the compiler can reduce the designs max frequency to achieve II=1. When the design have sub-loops with different number of iterations between them, the compiler usually can't pipeline the loops because the iterations can go out of order.

On NDRange kernel, loops work slightly different. Instead tof being pipelined, the loop can execute concurrently multiple iterations from different work items.

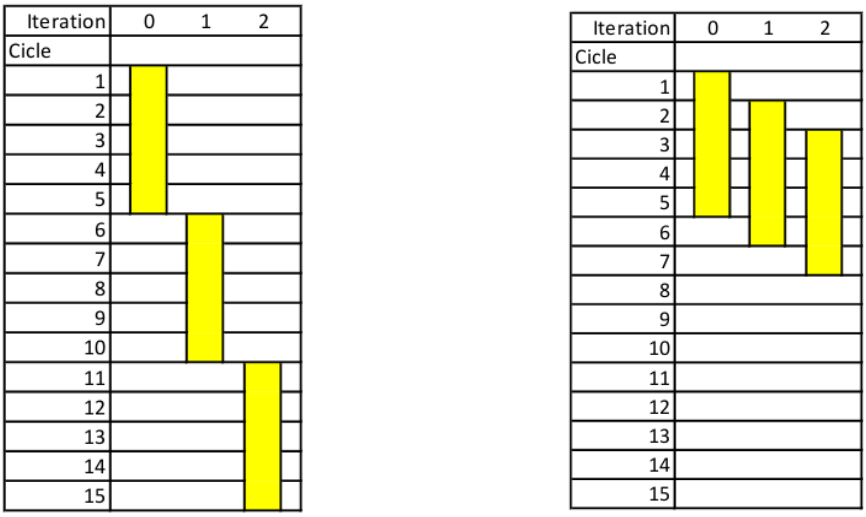


Figure 2.9: Left: loop iterations without loop pipelining. Right: loop iterations with loop pipelined

Work-items

Work-items are implemented as an integer id. A work-group is sent to a compute unit and the compute unit starts to send the work-items id to the pipeline. Work-items are independent between them so they don't need any control logic or feedback. That simplifies the hardware and avoid the loop structure overhead.

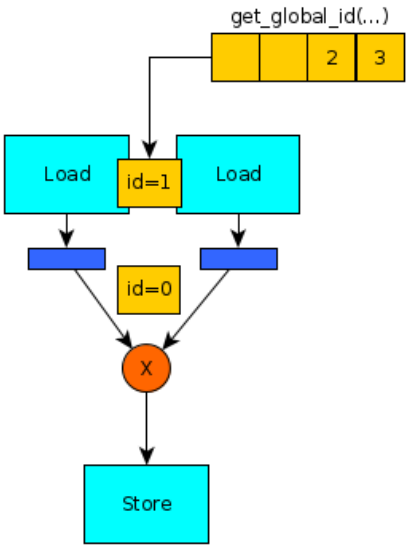


Figure 2.10: Work items pipeline

Compute units

When you define an OpenCL kernel in your code, you are defining and creating an instance of it. An instance of a kernel is a compute unit. This instances can be replicated using an attribute allowing to have more work-group running at the same time. Also, this allows to define more complex designs creating specialized units and communicate them by channels.

Channels

Channels are an OpenCL extension defined by Altera. Channels are a special type of variable structured as input/output that can only be read or written concurrently. They act like FIFOs and are useful to pass data between kernels without using global memory. They are translated to registers or BRAMs if the FIFO requires a large depth.

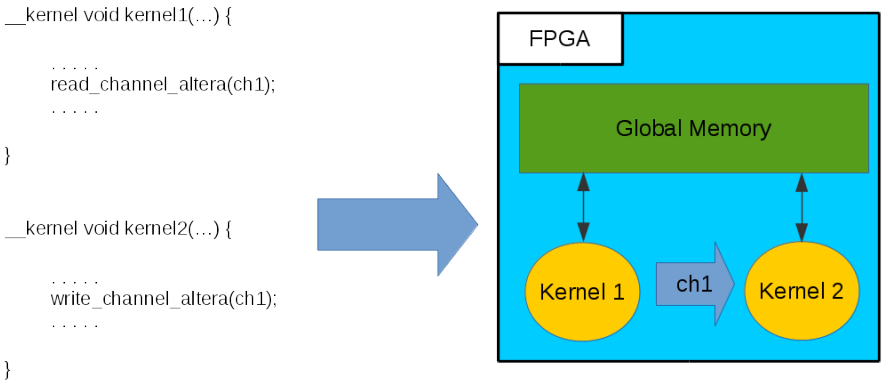


Figure 2.11: Channels on FPGA

Memory access

Intel OpenCL compiler looks at the memory access very carefully. It checks the index that we are using for our memory access searching for aligned access, sequential access, etc. When the memory access index has the form of $i + k$ where i is a variable with a constant increment and k is a constant, the compiler will generate streaming or semistreaming memory interfaces which are the fastest and most efficient ones. The main difference between them is that a semistreaming interface contains a cache and a streaming one no.

When the memory access gets more complicated, the compiler generates a burst interface. The memory access will be coalesced and acceded simultaneously whenever it is possible. With this interface, memory access has very large latency as they can be totally random access. To improve this, the compiler adds a small cache to the interface.

Finally, when the compiler is not sure that a memory access is aligned, a burst non aligned interface is generated. Similar to the previous one, but slower.

Environment limitations

The current OpenCL environment suffer from some limitations. We already mention one of them which is the FPGA area used for communications. This extra area for communications limits the usable area for our designs. Another limitation is the allocatable global memory. Our host have 64 GB of memory that theoretically are accessible by the FPGA because it is shared by the 2 devices,

but only 4GB can be allocated by the FPGA. This limits the size of our benchmarks.

Finally, the last limitation is that the OpenCL runtime is bounded to the thread that creates the OpenCL context. Only the bounded thread can use the OpenCL API so we can't use multiple threads to enqueue kernels, waiting executions to complete, etc.

2.5 Intel/Altera OpenCL design flow

The design flow of an OpenCL kernel for FPGA follows the following flowchart (Figure 2.12). We start writing the OpenCL code and compiling it for emulation to check it the correctness.

If the emulated code is correct, we can do an intermediate compilation. This generates Verilog from the OpenCL code and creates a report with some performance hints like the loops initialization interval. The report only produce this information for the single-thread kernels. NDRange kernels do not get this information since their loops are not pipelined, so you get very few information about them.

Finally, if you expect that the performance will be fine, you generate the bitstream. Notice that this last step takes many hours (between 6-8h in the best case, 9-11h when using nearly all board resources). So if you are optimizing a kernel, first optimize what you think that is the bottleneck, compile it (9-11h) and then test the bitstream to check is you where correct. Repeat this process over and over until you are satisfied with the performance. This is a very slow process with very little feedback from the compiler to know what is happening.

To evaluate the bitstreams, Altera provides a profiler. To use it, the bitstreams needs to be compiled with the –profile flag. This flag tells the compiler to add performance counters to the bitstreams. Only the memory access are instrumented so we can't get any other runtime feedback.

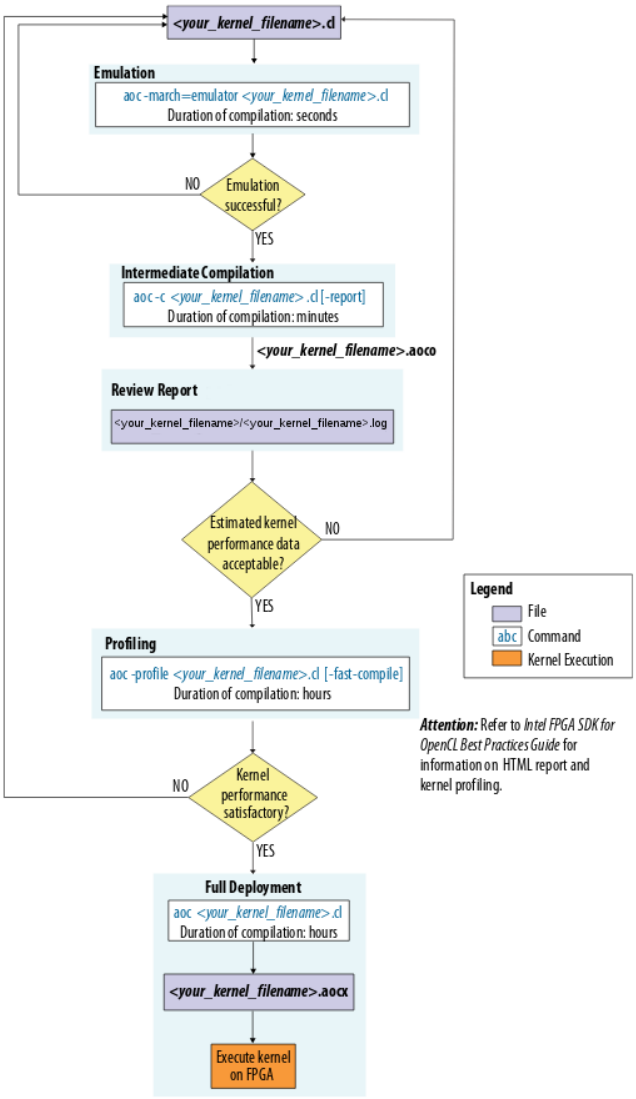


Figure 2.12: OpenCL FPGA design flow [2]

2.6 Automatic optimizations

The compiler provides pragmas and attributes to help us speedup the development. The most notorious are *__attribute__((num_simd_work_items))*, *__attribute__((num_compute_units))* and *#pragma unroll*

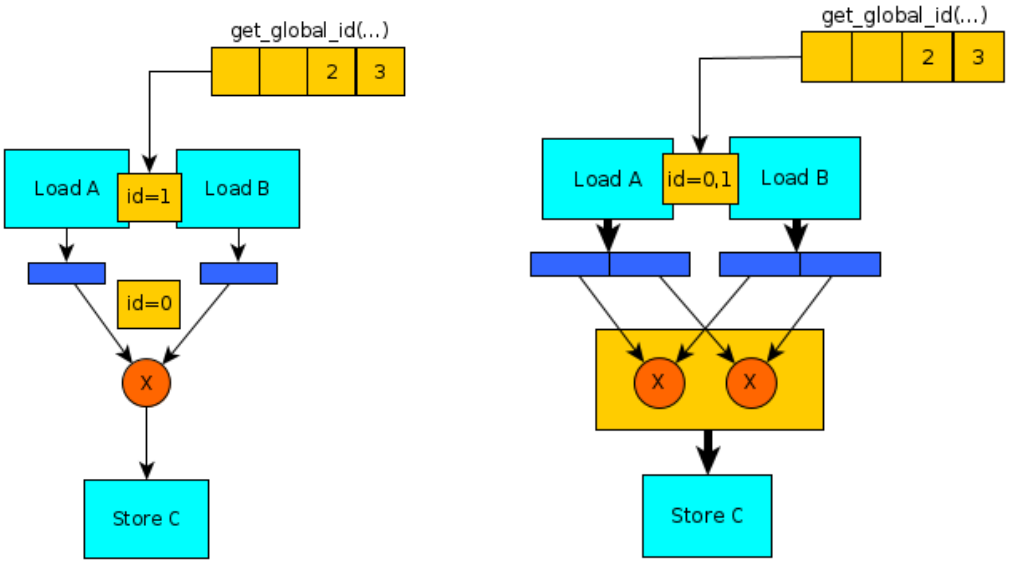
[2]Modified figure from Intel FPGA SDK for OpenCL https://goo.gl/Ybz9LV. Page 12

num_simd_work_items provide a way to make a wider pipeline. It applies to the NDRange kernels. Intuitively, what it does is to divide the number of work items in the work-group and split the work among the rest of the work items. For example, if we use a SIMD factor of 2 in a kernel with a work-group of 16 work-items, the kernel will execute 8 work items where the work item 0 will do the work of work-item 0 and 1, work-item 1 will do the work of work-items 2 and 3. To do that the pipeline needs to be wider.

```
1
2  __attribute__((reqd_work_group_size(4,1,1)))
3  __attribute__((num_simd_work_items(2)))
4  __kernel void foo( global int* A,
5                     global int* B,
6                     global int* C){
7
8      const size_t id = get_global_id(0);
9      C[id] = A[id]*B[id];
10
11 }
```

Figure 2.13: OpenCL code with SIMD attribute

Looking at figure 2.14a and 2.14b we can see the hardware resulting from the code at figure 2.13. As we can see, when we apply SIMD the pipeline gets wider and in practice what is doing is to execute 2 work-items simultaneously. The major improvement with this optimization is that is coalescing the memory access. In this example, we can see that, since the access are consecutive, we can fetch and store the data from the 2 work-items in a single petition, improving the bandwidth. If the memory access can't be coalesced, the compiler will create individual memory access but the rest of the attribute behavior keeps the same.



(a) Pipeline with SIMD factor of 1 (no SIMD)

(b) Pipeline with SIMD factor of 2

Figure 2.14

Another important attribute is *__attribute__((num_compute_units))*. This attribute replicates the kernel as many times as the developer needs. Replicating the kernel allows to distribute multiple work-groups between multiple compute units, computing them in parallel. Notice that, in contrast with the SIMD attribute, the memory petitions are increased because we have multiple unrelated memory petitions in parallel and they can't be coalesced.

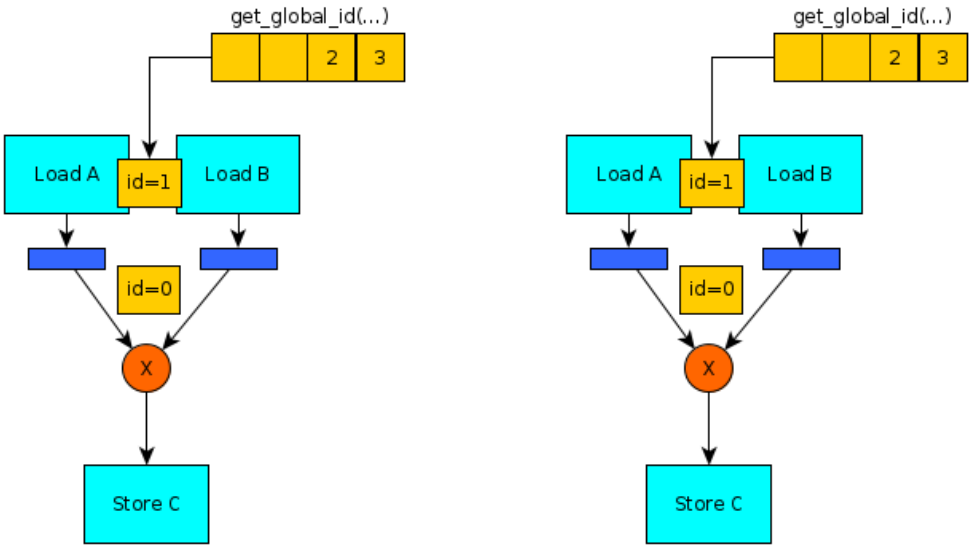


Figure 2.15: Replicated kernels with num_compute_units with factor 2

Finally, pragma unroll allows us to unroll automatically a loop, completely or not. In figure 2.17 we can see the pipeline resulting from figure 2.16 code. Has you can see it is pretty similar to the SIMD pipeline presented early (Figure 2.14b). This is because the compiler can detect memory access that can be merged which is the case. If we were accessing random memory locations the compiler would create non coalesced memory access (like in figure 2.18), increasing the memory access. Based on what we have described, unroll could provide results similar to SIMD in cases where the loop contains memory access and the idea is more or less the same: make the pipeline wider to do more work in parallel.

```
1
2  #define SIZE 16
3  __attribute__((reqd_work_group_size(1,1,1)))
4  __kernel void foo( global int* A,
5                     global int* B,
6                     global int* C){
7
8    #pragma unroll 2
9    for(int i = 0 ; i < SIZE ; ++i) {
10     C[i] = A[i]*B[i];
11   }
12
13 }
```

```
1
2  #define SIZE 16
3  __attribute__((reqd_work_group_size(1,1,1)))
4  __kernel void foo( global int* A,
5                     global int* B,
6                     global int* C){
7
8    for(int i = 0 ; i < SIZE ; i+=2) {
9      C[i] = A[i]*B[i];
10     C[i+1] = A[i+1]*B[i+1];
11   }
12
13 }
```

Figure 2.16: OpenCL code with loop unrolled 2 times (bottom). OpenCL code with pragma unroll with a unroll factor of 2 (top). This two codes are equivalent

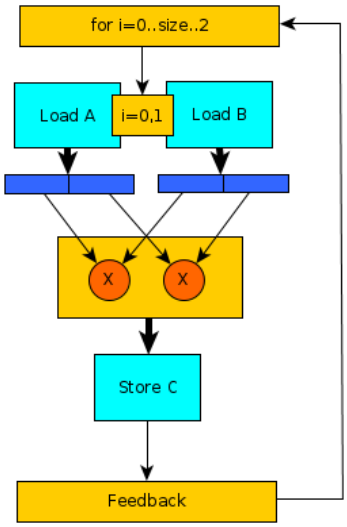


Figure 2.17: OpenCL code with pragma unroll and coalesced memory access

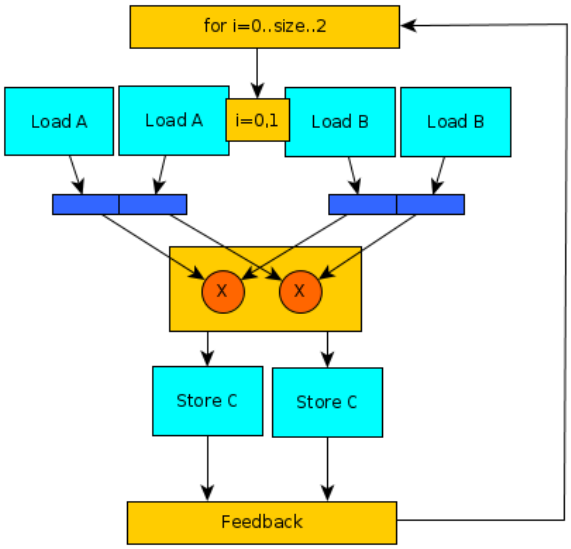


Figure 2.18: OpenCL code with pragma unroll and without coalesced memory access

Chapter 3

Related works

An OpenCL comparison between Xilinx FPGAs and NVIDIA GPUs is provided in [7]. They provide a set of benchmark ported to a Xilinx FPGA and an NVIDIA GPU using OpenCL. They present some weak points against a GPU: the FPGA is not affected by divergence and kernels can communicate between them using pipes avoiding the communication with global memory. In this cases, the FPGA can provide better performance per watt or even better performance.

[11] describe how a scientific program is ported to a FPGA using OpenCL. They show the porting to an Altera Stratix V and Xilinx Virtex 7 explaining the difference between them. They compare the optimized kernels with a GPU and a CPU with focus on the power consumption. They didn't provide a comparison with a shared memory architecture like ours and we think that the CPU that they used for the comparison is quite low end for a fair comparison. Probably the CPU can provide better performance and performance/watt

The authors of [8] present MOST(Method Of Splitting Tsunami) for tsunami simulations implemented on an FPGA. They evaluated on an Intel Arria 10 with PCIe and they use Intel HDL and compare them with a NVIDIA K20c and an AMD Radeon R9 280X getting speedup of 8.6x and 1.7x on performance and 17.2x and 7.1x on performance per watt. Unlike us, they evaluated on a Intel Arria 10 without shared memory and they used Intel HDL instead of Intel OpenCL, which is a little more high level than Intel HDL and doesn't require you to write a driver for communication like they did.

[6] presents an implementation of XSBench, a memory intensive Montecarlo simulation. They want to see how good performs an FPGA with irregular memory access. They evaluated on an Altera Arria 10. They saw that the FPGA implementation of XSBench achieves 50% higher energy efficiency than on Intel

Xeon 8-core CPU but the performance is 35% slower.

The authors of [5] evaluated a geografical system that requires intensive floating-point computations. The evaluation shows the energy efficiency of the single-precision kernel on the FPGA is 1.35X better than on the CPU and the GPU, while the energy efficiency of the double-precision kernel on the FPGA is 1.36X and 1.72X less than the CPU and GPU. They evaluated on an Intel Knight Landings, an NVIDIA Kepler and on an Intel Arria 10. They used Intel OpenCL for the FPGA. Unlike them, our evaluation is done on a shared memory architecture with an Intel Xeon and an Intel Arria 10.

Altera developers show in [2] that a PCIe Stratix V can get a 3x on performance agains a NVIDIA C2075 Fermi GPU and a 114x against a Intel Xeon W3690. They wrote a parameterizable fractal video compression algorithm that can run on multiple platforms and evaluate them. We evaluated our implementations on a shared memory architecture with an Intel Arria 10 which have more resources and doesn't have overhead for communications, and also evaluated our CPU codes using OpenMP or OmpSs which are more mature parallel models than OpenCL on the CPU.

Unlike GPU or CPU which are instruction interpreters, FPGA is a large network of components interconnected between them to generate a pipeline. The effort needed by the compiler to do the network routing is proportional to the resource utilizations. In [4] we can see how routing congestion or compiler problems for large resource utilization make the compiler unable to generate the bitstream. This is a problem that we suffer since we are trying to use the max amount of FPGA resource and that limits the amount of resources that we can use.

Chapter 4

Test cases design

In the next sections, we will describe the algorithms that we have ported to the FPGA and which optimizations have been performed. There are some common optimizations for all the kernels. We added a *restrict* attribute to the kernels inputs whenever we could. Also, when floating point is used, we use the –fp-relaxed and –fpc flags so the compiler can reorder floating point operations and regulate the precision so it can save resources and improve precision.

Experimental setup

For this work we used the following tools: For the OpenCL support on the FPGA we used Intel SDK OpenCL 16.0.2 which supports OpenCL 2.0. For our CPU codes we used GCC 6.3.0. OmpSs codes compiles with Mercurium but this compiler only does source-to-source for the pragma annotations and then call the same version of GCC. For our OmpSs codes we use the runtime Nanox 0.10 and the compiler Mercurium 2.0.0.

Our metrics are generated on a SoC with an Intel Xeon Processor E5-2600 v4 Family with an integrated Intel Arria 10. The two devices shares the memory.

The power metrics are generated using Quartus PowerPlay power analyzer. This tool generates a power consumption estimation analyzing the FPGA bit-stream.

4.1 NBody

Algorithm description

The NBody algorithm computes the force and the velocity between the particles from a particle system, simulating how the system evolves over time.

Algorithm 1 NBody algorithm pseudocode

```
procedure NBODY(K)
    for all i ∈ K do
        force_acc ← 0
        for all j ∈ K do
            if i ≠ j then
                force_acc ← CALCULATEFORCE(i, j)
            end if
        end for
        new_velocity ← COMPUTENEWVELOCITY(force_acc)
        new_position ← COMPUTENEWPOSITION(force_acc)
        output_position[i] ← new_position
        output_velocity[i] ← new_velocity
    end for
end procedure
```

Analysis and optimizations

We started with an OmpSs@OpenCL GPU implementation of NBody. The OpenCL kernel is designed as a 1 dimension ND-range with as many thread as particles (the work-items doesn't share local variables so we don't care about the work-group size). Each thread computes in parallel the force between a particle and the rest of the particles and updates the particle state (velocity and position). Each call to the kernel, computes a timestep. We write a CPU implementation with AVX2 and OmpSs to compare it.

Looking at the memory accesses

The initial implementation was very slow compared with the CPU. We test it with the profiler to check the memory cache efficiency. The hit rate was 99% so the memory accesses are not the issue. Since the profiler can't report anything else we are a bit blind here, but as there's no memory bottleneck it must be a compute bottleneck.

Saturating the resources

We have plenty of resources available so we can try to apply the optimizations described early. The program reads sequentially arrays from memory which means that if we apply SIMD or unrolling, the memory access will be coalesced. Coalesced memory accesses is the best way to exploit bandwidth so, we tried both optimizations.

As we can see in figure 4.1, the performance improve dramatically, being faster than the CPU. We didn't include the FPGA version without optimizations for chart readability (you can seen it at table 4.1. FPGA Nbody without optimizations its always slower than the CPU or FPGA with optimizations. The unroll approach performed better than the SIMD, probably because in the unroll approach the compiler is coalesing the memory access, just like in the SIMD approach, but also is doing more loop iterations simultaneously since the unroll factor is greater than the SIMD one.

#Particles	**CPU**	**FPGA raw**	**FPGA SIMD**	**FPGA unroll**
16384	1.2035	10.9785	0.8582	0.4672
32768	4.8112	43.3901	3.3563	1.7983
65536	19.2396	172.6360	13.3677	7.1090

Table 4.1: Time in seconds for NBody FPGA versions

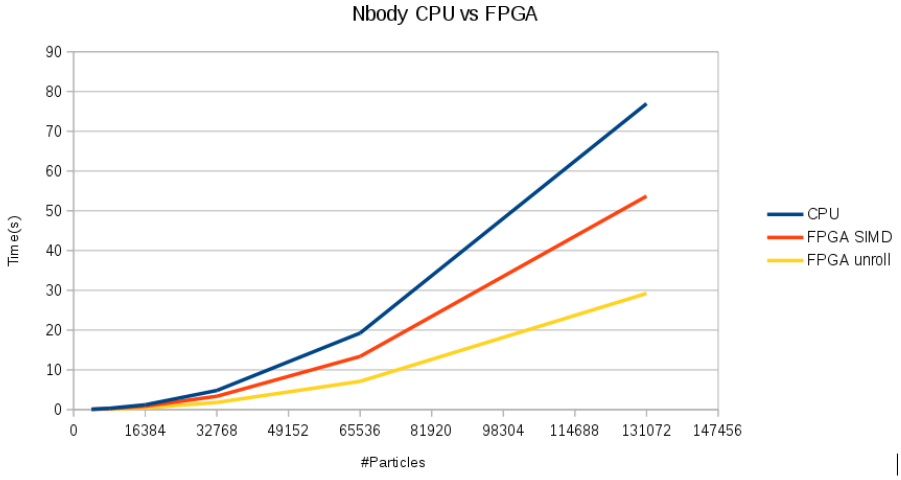


Figure 4.1: Nbody optimized results

Work groups performance impact

We still can try another approach which is to unroll the loop using a 2D NDRange kernel distributing the iterations between multiple work-items. We redesigned our kernel to a 2 dimensions NDRange, where the X coordinate determinate which particle it calculates. All the threads in the same X coordinate but with a different Y computes the same particle and split the loop work between them.

With this design we need a reduction to merge the computed force. This adds another loop that can be unrolled but reducing the available resource for the main loop. Atomic operations are not available. We also have another problem which is that threads can hit the barriers in different order, so the compiler limit the number of concurrent work-groups to 2, limiting performance.

In figure 4.2 we can see that the approach didn't work well. Since we are complicating the code adding more complex things like reductions, this is making the resulting hardware more inefficient. Also, adding more threads probably is adding delays and more threading management on the main loop, making it more inefficient.

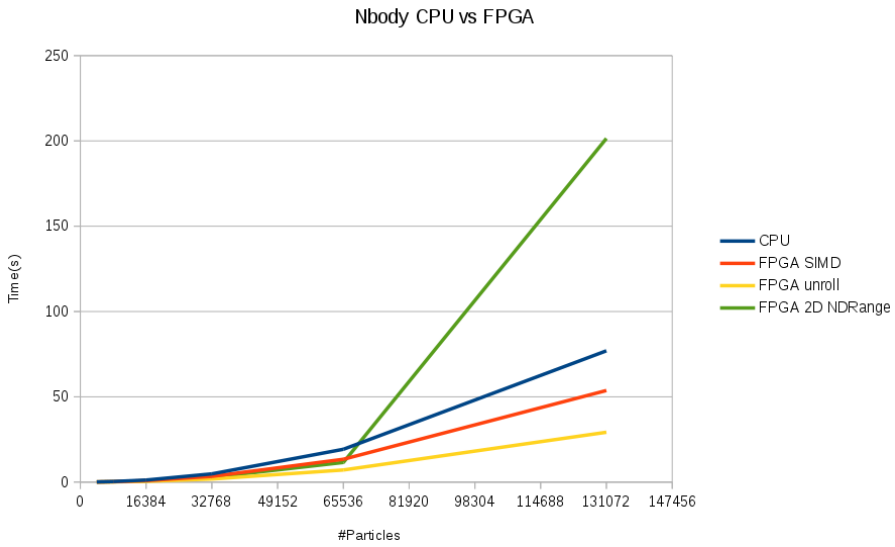


Figure 4.2: Nbody optimized results with 2D NDRange

If we talk about energy, the results as been successful. As we can see in figure 4.3, optimized NBody scales very well in energy consumption. The raw version, which we didn't included, consumes more than the CPU, it is too slow. The consumed energy in the FPGA version does not include the CPU consumption

since the CPU is idle waiting the kernels to finish. It is true that the CPU needs to be there but we always need a CPU and in this case it can do other tasks while is waiting for the FPGA to finish so adding the CPU consumption to the FPGA is not fair.

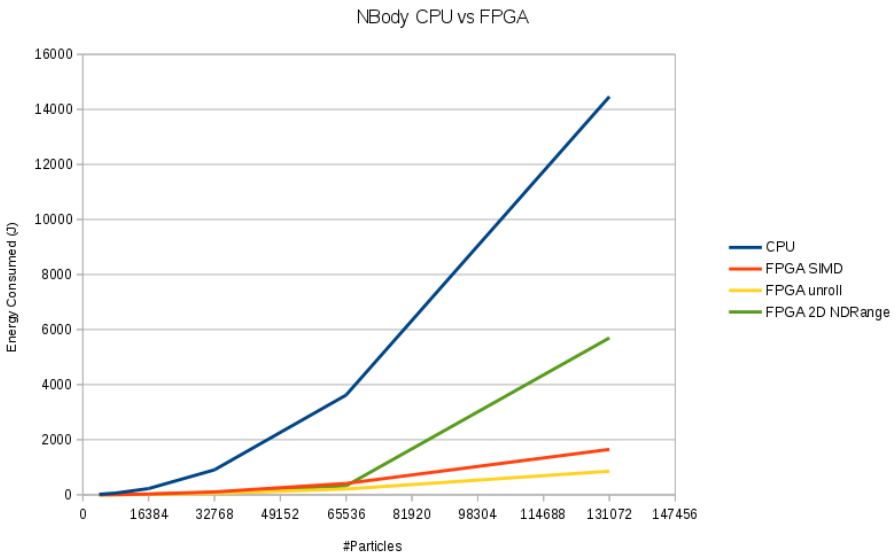


Figure 4.3: NBody energy consumption

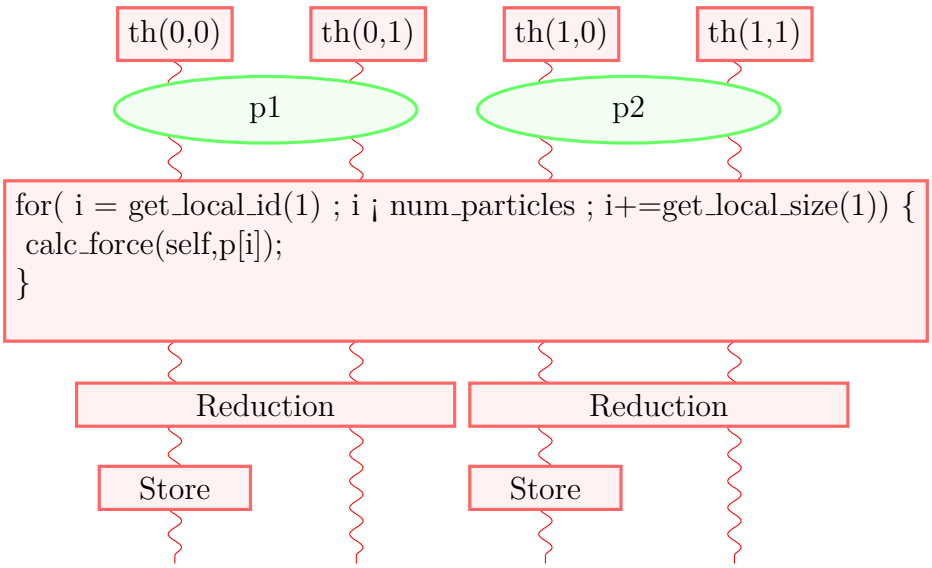


Figure 4.4: NBody - work-groups X dim determinate the particle. Work-groups Y dim cooperate to calculate the particle force

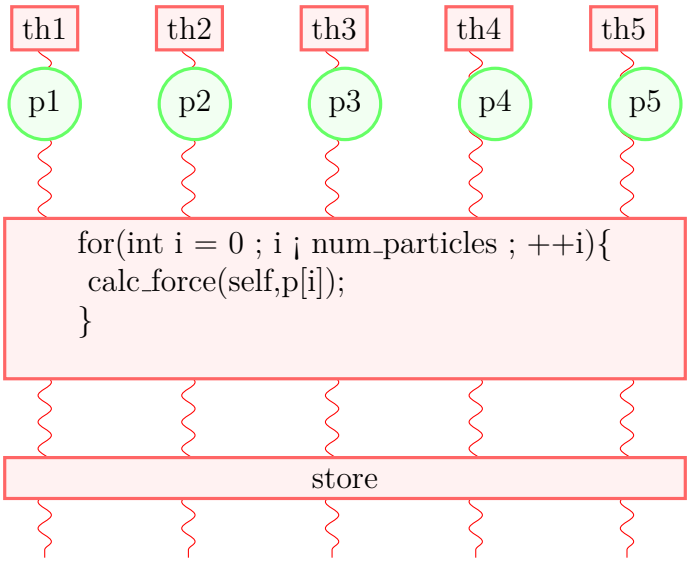


Figure 4.5: NBody OpenCL basic implementation

4.2 Cholesky

Algorithm description

The Cholesky decomposition generates a lower triangular matrix and its conjugate using a Hermitian positive-definite matrix. The product of the 2 matrix is the provided Hermitian matrix.

$$A = LL^T = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{bmatrix} \tag{4.1}$$

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2} \tag{4.2}$$

$$L_{ij} = \frac{1}{L_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} \right) \quad for\ i > j \tag{4.3}$$

Figure 4.6: Cholesky algorithm

Since the access pattern is not cache-friendly for big matrix, it is likely to run the algorithm by blocks. Cholesky is only computed in the diagonal blocks and

every time we compute a block, the changes need to be propagated to the rest of the matrix.

Algorithm 2 Cholesky blocked algorithm

procedure CHOLESKY_BLO(A,bs)
  for $j = 0 : n - 1$ **do**
    $b \leftarrow min(n - j, bs - 1)$
    $A_{j+b:j+b-1,j:j+b-1} \leftarrow CHOLESKY(A_{j:j+b-1,j:j+b-1})$
    $A_{j+b:n,j:j+b-1} \leftarrow A_{j+b:n,j:j+b-1} * A^{-T}_{j:j+b-1,j:j+b-1}$ ▷ TRSM
    $A_{j+b:n,j+b:n} \leftarrow A_{j+b:n,j+b:n} - TRIL(A_{j+b:n,j:j+b-1} * A^{T}_{j+b:n,j:j+b-1})$ ▷ Triangle lower GEMM/SYRK
  end for
end procedure

Analysis and optimizations

We start writing a CPU version of blocked Cholesky factorization using OmpSs and Intel MKL to compare with our OpenCL version. A CPU blocked cholesky contains 4 MKL routines: Cholesky, TRSM, GEMM and SYRK. We wrote OpenCL versions of this kernels so we can run the blocked Cholesky algorithm on the FPGA.

The 4 kernels fit on the FPGA, but they cover nearly all the area leaving very few space for optimizations. Since SYRK is a particular GEMM case where the 2 input pointers are the same, having 2 different kernels will lead to a worst performance than having one kernel optimized using the area of the other one so will remove this. Even with this, the initial performance results where very bad as we can see in table 4.7.

Cholesky and TRSM kernels are very problematic since they have data dependencies in the loops and many subloops. Data dependencies increase the II(initialization interval) and subloops, in some cases, doesn't allow the compiler to pipeline the loop since the iterations between the loops can get out-of-order. GEMM is not having this problems and looks relatively easy to optimizes since there are no memory dependencies inside the loop and the memory access can be vectorized. Notice that we are not computing a regular GEMM, we are computing $A * B^T$ which means that we can read B by rows.

Matrix size	**CPU BS=128**	**CPU BS=256**	**CPU BS=512**	**FPGA 3 kernels**
2048	0.0216	0.0292	0.03255	1.9045
4096	0.0552	0.0475	0.07971	22.8588
8192	0.3382	0.2338	0.2821	723.1120

Figure 4.7: Performance in seconds Cholesky with 3 kernels vs CPU

Looking at the CPU version, the most common kernels are GEMM. The GEMM computes $A * B^T$ so we can read the matrix sequentially. We expect that the GEMM can exploit well FPGA resources so we decided to use the FPGA GEMM and run the rest of the kernels on the CPU as MKL routines.

We started with a single thread kernel implementation. The code looks like a basic matrix multiply but since we are computing $A * B^T$ we read B by rows.

```
1  for(i = 0 ; i < BS ; ++i) {
2    for(j = 0 ; j < BS ; ++j) {
3      for(k = 0 ; k < BS ; ++k) {
4        C[i*BS+j] = b*C[i*BS+j] + a*A[i*BS+k]*B[j*BS+k];
5      }
6    }
7  }
```

Figure 4.8: Single thread matrix multiply - no optimizations

Memory access are a bit more complicate than Nbody, so accessing directly inside this loops leads to inefficient memory access, more area usage and complex scheduling that increase the loops II(initialization interval). We loaded A and B inside BRAMs to solve this issues. To make this efficient, we load 16 floats at time to maximize the bandwidth usage. We can see the optimized code at figure 4.9

```
1   __local REAL localA[BS][BS];
2   __local REAL localB[BS][BS];
3
4   for(size_t i = 0 ; i < BS*BS ; i+=16){
5     REAL16 tmp1 = A[i/16];
6     REAL16 tmp2 = B[i/16];
7     localA[i/BS][i%BS+0]  = tmp1.s0;
8     localA[i/BS][i%BS+1]  = tmp1.s1;
9     localA[i/BS][i%BS+2]  = tmp1.s2;
10    localA[i/BS][i%BS+3]  = tmp1.s3;
11    localA[i/BS][i%BS+4]  = tmp1.s4;
12    localA[i/BS][i%BS+5]  = tmp1.s5;
13    localA[i/BS][i%BS+6]  = tmp1.s6;
14    localA[i/BS][i%BS+7]  = tmp1.s7;
15    localA[i/BS][i%BS+8]  = tmp1.s8;
16    localA[i/BS][i%BS+9]  = tmp1.s9;
17    localA[i/BS][i%BS+10] = tmp1.sa;
18    localA[i/BS][i%BS+11] = tmp1.sb;
19    localA[i/BS][i%BS+12] = tmp1.sc;
20    localA[i/BS][i%BS+13] = tmp1.sd;
21    localA[i/BS][i%BS+14] = tmp1.se;
22    localA[i/BS][i%BS+15] = tmp1.sf;
23    localB[i/BS][i%BS+0]  = tmp2.s0;
24    localB[i/BS][i%BS+1]  = tmp2.s1;
25    localB[i/BS][i%BS+2]  = tmp2.s2;
26    localB[i/BS][i%BS+3]  = tmp2.s3;
27    localB[i/BS][i%BS+4]  = tmp2.s4;
28    localB[i/BS][i%BS+5]  = tmp2.s5;
29    localB[i/BS][i%BS+6]  = tmp2.s6;
30    localB[i/BS][i%BS+7]  = tmp2.s7;
31    localB[i/BS][i%BS+8]  = tmp2.s8;
32    localB[i/BS][i%BS+9]  = tmp2.s9;
33    localB[i/BS][i%BS+10] = tmp2.sa;
34    localB[i/BS][i%BS+11] = tmp2.sb;
35    localB[i/BS][i%BS+12] = tmp2.sc;
36    localB[i/BS][i%BS+13] = tmp2.sd;
37    localB[i/BS][i%BS+14] = tmp2.se;
38    localB[i/BS][i%BS+15] = tmp2.sf;
39  }
40
41  for(size_t i = 0 ; i < BS*BS; ++i){
42        REAL sum = 0;
43        const REAL tmpc = C[i];
44         #pragma unroll
45        for(size_t k = 0 ; k < BS ; ++k){
46            sum += localA[i/BS][k]*localB[i%BS][k];
47        }
48
49        C[i] = beta*tmpc+alpha*sum;
50  }
51 }
```


Figure 4.9: Single thread matrix multiply with optimizations

In the compute loops, the inner loop can be fully unrolled so we can do all floating point operations in parallel. Unrolling the second loop leads to a very complicated scheduling since the design needs a lot more read ports for the BRAMS so it is not really a feasible option. We can't remove the loop unrolling it but we can merge the 2 remaining loops. The more external loop have a II=2 for some compiler limitation, so merging them will remove this issue. The last thing that we can do is to unroll the merged loop until the resources are saturated.

We have also written a threaded version. In this version, each work-group compute a row of points. Each thread loads one of the elements from row A on local memory and then iterate on a B. B accesses are efficient because the compiler is generating a cache and access are sequential to the same position. We also unroll completely the only loop in the kernel to get rid of non pipelined areas.

```
1
2       __local REAL localA[BS];
3       const size_t tid = get_local_id(0);
4       const size_t grpid = get_group_id(0);
5       const size_t offA = BS*grpid; //offset A
6       const size_t offB = BS*tid; //offset B
7       const size_t offC = BS*grpid; //offset C
8
9       //load A
10      localA[tid] = A[offA+tid];
11      barrier(CLK_LOCAL_MEM_FENCE);
12
13      REAL dotcalc = 0;
14      #pragma unroll
15      for(size_t i = 0 ; i < BS ; ++i){
16          dotcalc += localA[i]*B[offB+i];
17      }
18
19      C[offC+tid] = C[offC+tid]*beta + alpha*dotcalc;
```

Figure 4.10: GEMM computing one row per work-group

Considering we wanted to test how good it performs with a single work-group, we have written a thread version where a single work-group computes an entire block. The threads are in a single dimension, so we can write the memory access in the form of $i + k$, making it more efficient. All the data is loaded to BRAMS since all the data is shared between the same work-group,

saving bandwidth and improving speed. We unrolled completely the only loop in the kernel. Always remember that loops in NDRange kernels are not pipelined, so removing them is always a win.

```
1   __local REAL localA[BS][BS];
2   __local REAL localB[BS][BS];
3
4   const size_t id = get_global_id(0); //0..BS*BS
5   const size_t tid0 = id/BS;
6   const size_t tid1 = id%BS;
7
8   const size_t index = id;
9
10  //load A
11  localA[tid0][tid1] = A[index];
12  localB[tid0][tid1] = B[index];
13
14  barrier(CLK_LOCAL_MEM_FENCE);
15
16  REAL dotcalc = 0;
17  #pragma unroll
18  for(size_t i = 0 ; i < BS ; ++i){
19      dotcalc += localA[tid0][i]*localB[tid1][i];
20  }
21
22  C[index] = C[index]*beta + alpha*dotcalc;
```

Figure 4.11: GEMM computing one block per work-group

Looking at the figure 4.12 We can see how it performs each optimization compared against the implementation with 3 kernels. The benchmarks with block size of 128x128 end at matrix size of 4096x4096 since the 8192x8192 execution takes hours. We can see that our optimizations improved the performance, but what really improves the execution time is the block size. Using a block size of 256x256 instead of 128x128 always drops the execution time from the order of 20 seconds to aprox 1-2 seconds.

Block size	FPGA O1	FPGA O2	FPGA O3	MKL
128	0.9382ms	0.1458ms	0.1469ms	0.1860ms
256	0.1929ms	5.0486ms	0.6768ms	0.8414ms

Table 4.2: Performance table from the GEMM kernels and MKL. O1 is the single-thread version, O2 is the one work-group per row version and O3 is the one work-group per block optimization

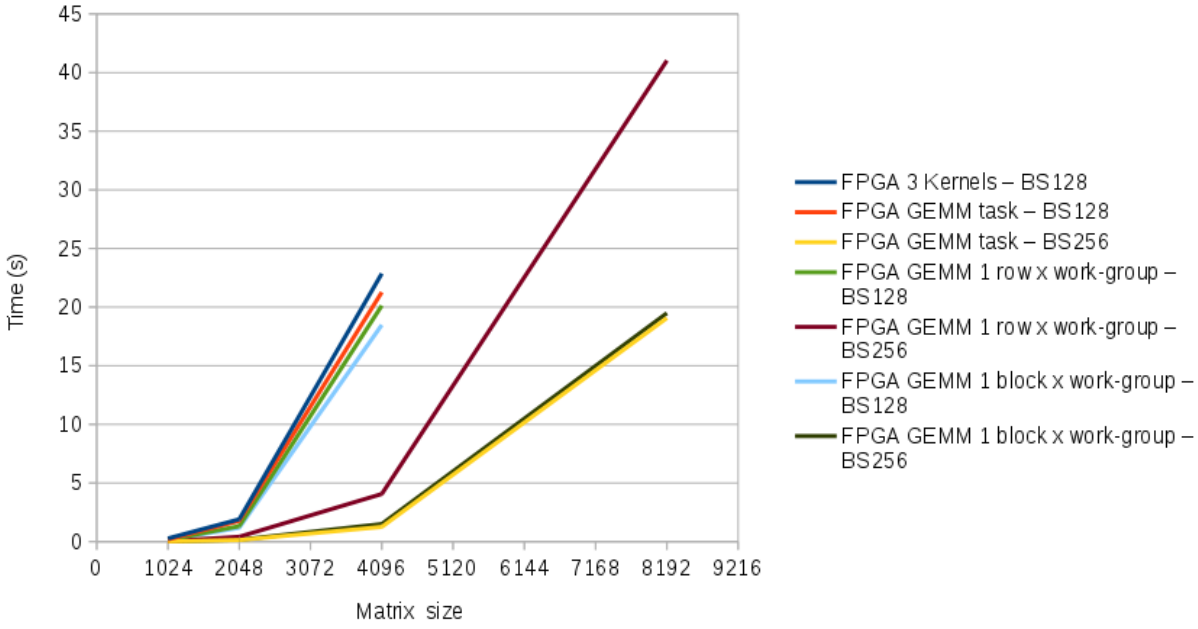


Figure 4.12: Time comparison between GEMM FPGA versions

We start looking at the issue running the kernels individually. As we can see in the table 4.3 with both block size, 2 of 3 FPGA kernel performs better than MKL. So where is the issue?.

We generate a trace from the parallel execution using Extrae which is supported by OmpSs. Comparing the figures 4.13 and 4.14 we can see a great unbalance between the threads in the OpenCL version which doesn't happens in the SMP version. In the SMP version, we have 24 threads that can do 24 GEMM simultaneously and, in the OpenCL, only the FPGA (thread 2 in the figure 4.14) can only do one GEMM simultaneously. As we can see in the table 4.3, kernels are slower with bigger block size, so the performance difference comes from the number of blocks. Since with bigger block size the number of blocks is reduced, the amount of work that the FPGA needs to do is less and the performance drop

comes later.

Matrix size	**CPU BS=128**	**CPU BS=256**	**CPU BS=512**	**FPGA BS=128**
2048	0.0216s	0.0292s	0.0325s	1.9045s
4096	0.0552s	0.0475s	0.0797s	22.8588s
8192	0.3382s	0.2338s	0.2821s	723.1120s

Table 4.3: Performance table between CPU and Cholesky on FPGA with 3 kernels

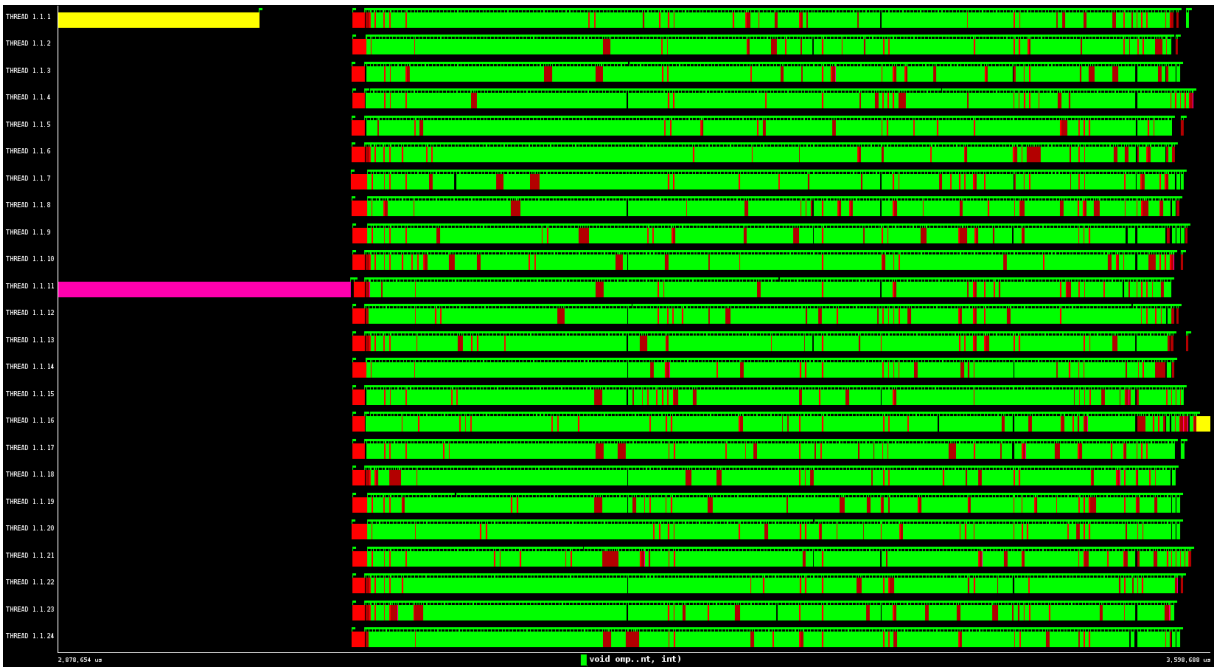


Figure 4.13: Extrae trace from Cholesky SMP

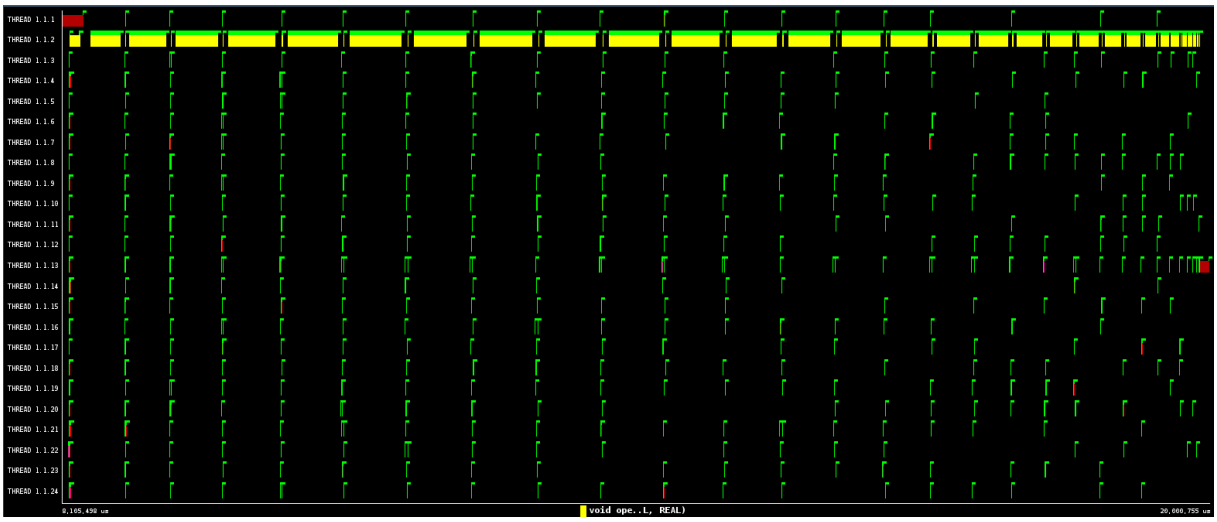


Figure 4.14: Extrae trace from cholesky OpenCL

Finally, as we see in figure 4.15 the energy consumed by the program does not benefit from the FPGA. Since the CPU is involved in the computations, we also add its consumption to the results which are the mostly the 80% of the power consumption.

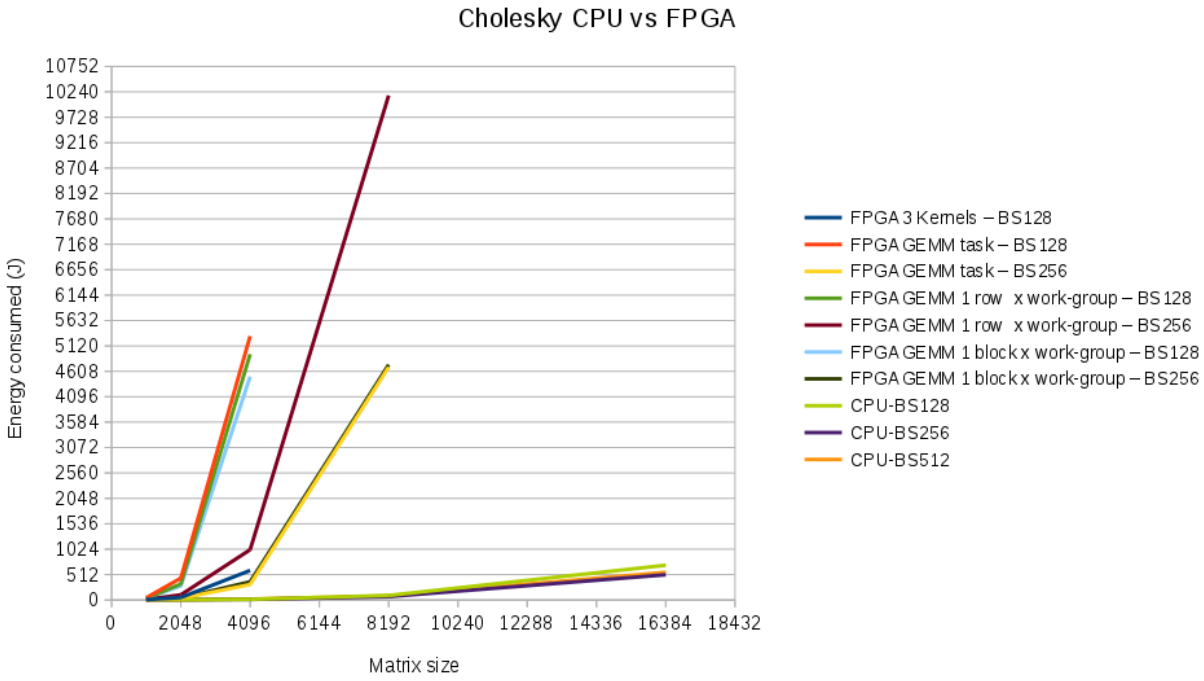


Figure 4.15: Energy consumption from Cholesky SMP and FPGA

Systolic matrix multiply

We have a matrix multiply that performs well but is far from the peak performance of the device. To get higher performance, you need to design a systolic array. A systolic array is a network that interconnects a group of nodes that performs some task. In this case, each node represents a cell from the output matrix so they take 2 values, multiply them and add them to their stored value. The nodes also send forward the values that they receive.

We didn't implemented this kind of network, but Altera did it on an Arria 10 FPGA [10] and they achieve 1 Teraflop. Notice that this paper describe an implementation on PCIe FPGA where the communication isn't implemented on the FPGA, so they have a lot more area to use. Also, the frequencies that they claim are higher than the ones that can be achieved in our FPGA. An empty kernel in our FPGA can get a frequency about 210MHz so provably the frequency is limited by the communication implemented on the FPGA.

4.3 CFD

Algorithm description

The CFD solver is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow.

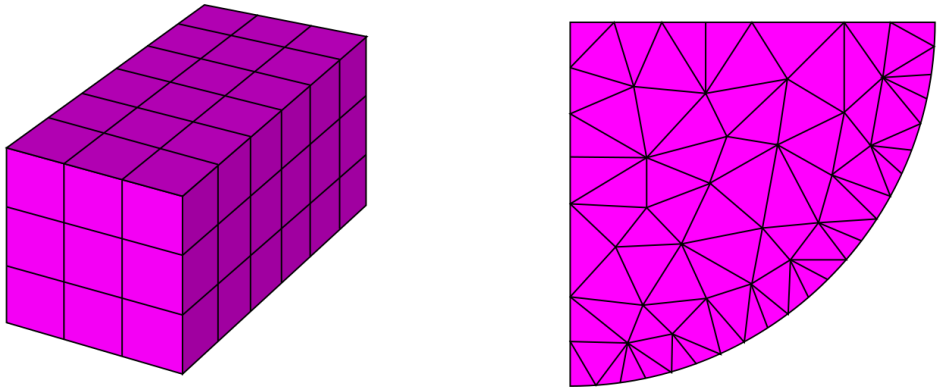
$$\frac{d}{dt}\int_{\Omega}\mathbf{u}d\Omega+\int_{\Gamma}\mathbf{F}\cdot\mathbf{n}d\Gamma=0, \tag{4.4}$$

$$\mathbf{u}=\begin{Bmatrix}\rho\\ \rho v_x\\ \rho v_y\\ \rho v_z\\ \rho e\end{Bmatrix}, \underline{\mathbf{F}}=\begin{Bmatrix}\rho v_x & \rho v_y & \rho v_z\\ \rho v_x^2+p & \rho v_x v_y & \rho v_x v_z\\ \rho v_y v_x & \rho v_y^2+p & \rho v_y v_z\\ \rho v_x v_x & \rho v_x v_y & \rho v_z^2+p\\ v_x(\rho e+p) & v_y(\rho e+p) & v_z(\rho e+p)\end{Bmatrix} \tag{4.5}$$

$$p=(\gamma-1)\rho\left[e-\tfrac{1}{2}\|\mathbf{v}\|^2\right]. \tag{4.6}$$

$\rho, v_x, v_y, v_z, e, p, \gamma$ denote, respectively, the density, x, y, z, velocities, total energy, pressure and ratio of specific heats.

The space is divided in an unstructured grid that requires indirections to access it.



(a) Structured grid (b) Untructured grid

Figure 4.16: Grid types [1]

Analysis and optimizations

The initial implementation is the Rodinian [1] OpenCL version. Our first problem was that the kernels barely fit on the FPGA. The program was very optimized to avoid communications with the CPU so it contains many trivial kernels just to move data. We don't have transmission problems since we are on shared memory so we removed all the trivial kernels and we left a single kernel which is the one that computes the algorithm. This leaves us with some more area to optimize the real work. Some of the memory reference that the kernel receive only contains 3 constant floats. This force the compiler to create memory interfaces (which are expensive) when this floats can be kernel parameters providing a faster kernel and using less area.

With all the area that we gain we can look at multiple optimizations. One can be to apply vectorization the code, but this is no feasible. Memory access mostly read columns and indirect access so the access can't be coalesced limiting the bandwidth to feed a wider pipeline . The other optimization is to unroll the loops. The code contain 2 big loops with very few iterations that do all the work. We can unroll them so we eliminate all the loops in our code.

[1] https://goo.gl/3ckKDe

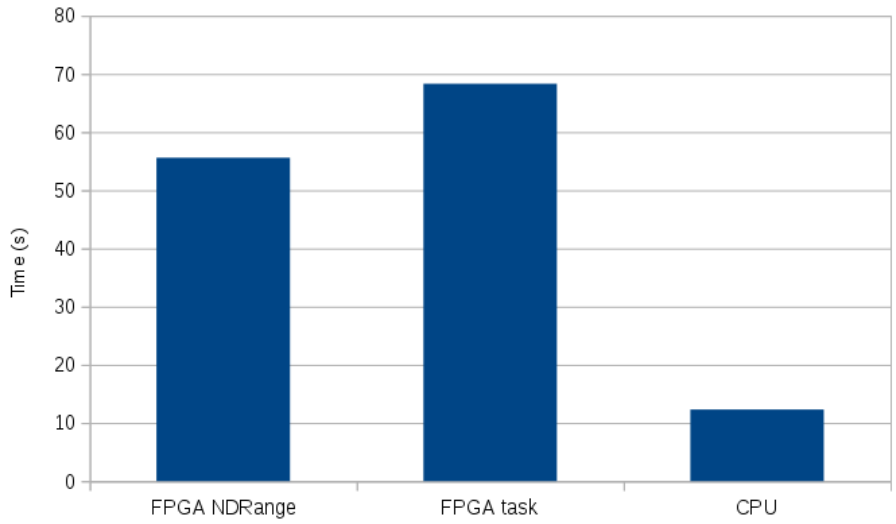


Figure 4.17: Comparison between CFD implementation on has an NDRange, task(single-thread) and OpenMP CPU

The kernel that was provided was an NDRange kernel but we also written a single-thread version so we can compare the performance of the 2 versions. The CPU version is the OpenMP version provided by Rodinia [1] . We can see in the figure 4.17 that NDRange peforms 22% faster than the task, but compared with the CPU they perform far slower. Looking at the profiler (figure 4.18) we see that the kernel is stall like 75% of the time in indirect memory access. The bandwidth is not at it full capacity so what we think is that the latency of the index is making the kernel get stall here, waiting for the index. The other possible reason is that, since the accesses are nearly random, we are always paying the memory latency making the kernel slow.

As indirect memory accesses are part of the algorithm we can't improve any better.

Line #	Source: Kernels.cl	Attributes	Stall%
206	density_nb = variables[nb + VAR_DENSITY*nelr];	0: __global{MEMORY},read	0: 71.52%
207	momentum_nb.x = variables[nb + (VAR_MOMENTUM+0)*nelr];	0: __global{MEMORY},read	0: 72.72%
208	momentum_nb.y = variables[nb + (VAR_MOMENTUM+1)*nelr];	0: __global{MEMORY},read	0: 71.19%
209	momentum_nb.z = variables[nb + (VAR_MOMENTUM+2)*nelr];	0: __global{MEMORY},read	0: 72.35%
210	density_energy_nb = variables[nb + VAR_DENSITY_ENERGY*nelr];	0: __global{MEMORY},read	0: 51.13%

Figure 4.18: CFD profiler trace

Also, since we moved part of the code to the CPU, the energy consumption is higher since the CPU is involved. As we can see in the figure 4.19, the energy consumption comes mostly from the CPU and nearly all is waiting the FPGA to finish as the FPGA design is much slower than the CPU.

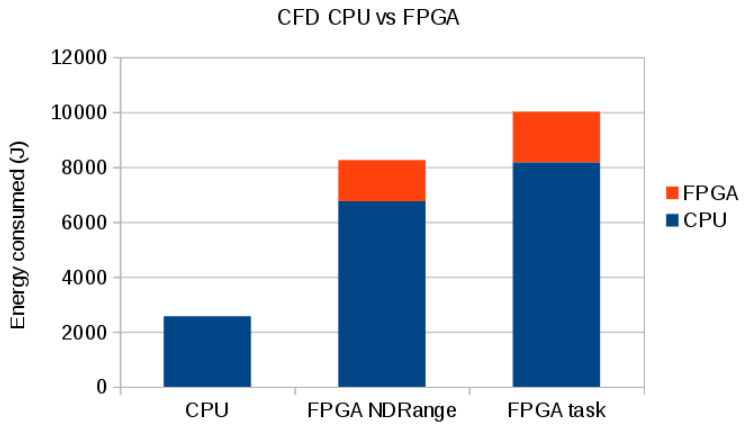


Figure 4.19: CFD energy results. Each color represents the contribution of the device (FPGA or CPU) to the tests

4.4 Mergesort

Background

Initially, we started porting Hybridsort provided by the Rodinian benchmark [1] . The program is designed to split the work into buckets using a histogram. Then it sorts the buckets and it finally merge them. The first tests with this design show a really bad performance, This design have some indirect memory accesses which are extremely costly on the FPGA (As we have seen on the CFD) and they are difficult to remove. For this reasons we decided to redesign the kernel so it is more suitable for the FPGA.

Algorithm description

The basic mergesort algorithm splits recursively an array until the array segment is trivial to sort. Then, it starts to go back, merging the sorted segments. Since the segments are already sorted, the cost of merging has linear cost. Mergesort can be implemented on hardware as a comparators tree. Our implementation start at the merge point. Each node receives a list of sorted floats. We will give more details in the next section.

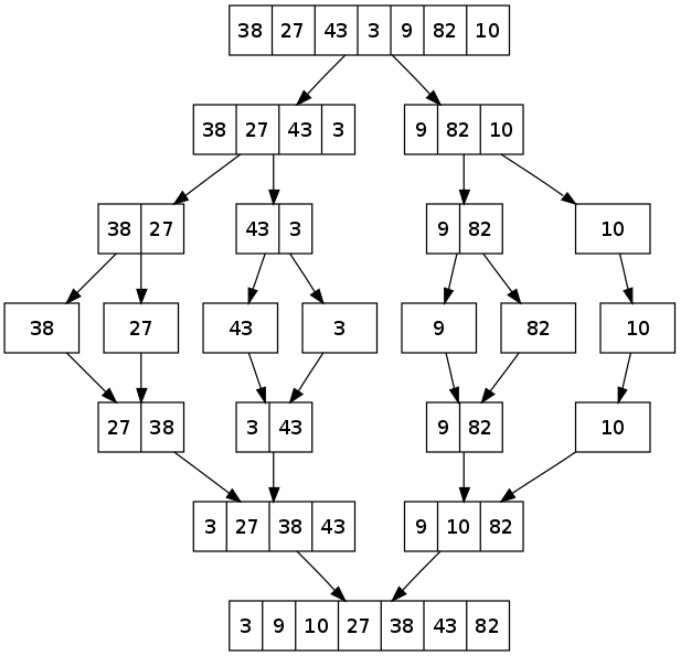


Figure 4.20: Mergesort algorithm

Analysis and optimizations

To implement our design, we arrange a set of OpenCL kernels that will act like a comparators tree. The development of this scheme was very time consuming: Since each node communicate with channels, the design could get stuck on some points (and it did it very often). Channels can't be emulated and since the kernel doesn't finish we don't have any feedback. Also, as we want to make a big tree, and defining with OpenCL tools is really complicated, we generated this code with a script. The script generates a comparator unit kernel and it copies it the number of times that is necessary, defining the necessary channels and connections in the process. Writing that script was non-trivial since we never know whether a bug comes from the script or the design. In the next sections, we will describe each node of the design. All this kernels are single-thread kernels since is far easier to work with channels with a single-thread kernel and also we didn't find any node where using NDRange makes sense.

Input kernel

The design sorts arrays of N floats where N is the number of input kernel compute units. The input kernel reads sequentially a number of floats starting at a provided position. Early versions have only one input kernel that reads linearly

the memory data and distributed it in a round-robin way to the nodes, but the kernel got stuck. What was happening here was that the input kernel was waiting on some channel that was full and the nodes weren't draining it because they need the values from channels that are empty and they weren't filled. So we changed to multiple units that fetches the values. We could have modified the single unit design to not get stuck but that was inefficient as the memory access patters get more complex and the generated memory interface becomes more inefficient. If we have added only one kernel, the compiler will use a *burst cached* memory interface since the memory accesses aren't sequential. Also, it requires a BRAM for the cache and, as the channels between the kernels are stallable, we need additional control logic to avoid stall all the design.

With N compute units, the memory interface is a stream. Fast and cheap (doesn't require a cache). Also there is no stalling management since we only have to worry about the channel that the kernel is feeding. Stalling my channel doesn't imply that I'm starving the other channels as other compute units are feeding them.

Comparator unit

The comparator unit reads from the channels, compare the readed values and the smallest is written into the output. The bigger value is stored for the next iteration. Since we retain one of the values, we need to know the smallest so we know from which FIFO we need to read. This increments the initialization interval of the loop but is part of the algorithm and can't be avoided.

Also, this unit have some logic for the list size that is ordering. Every comparator unit receives the size of the ordered list that is ordering, so each node knows when they will not receives more elements from a channel and can write the remaining values.

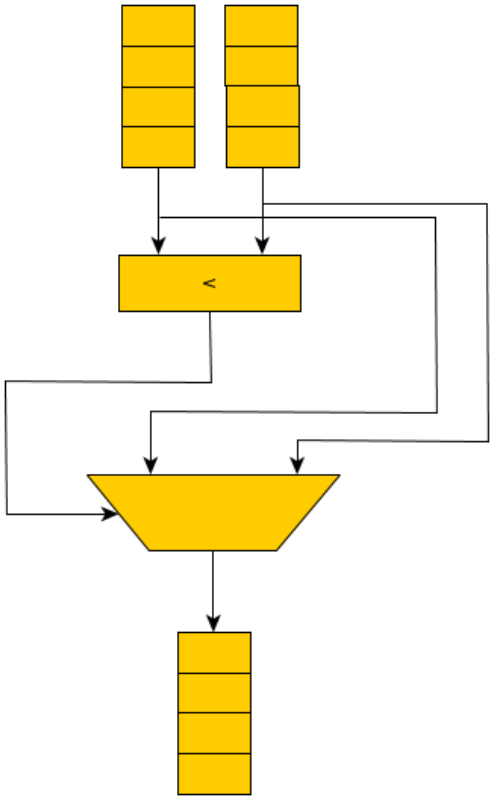


Figure 4.21: Comparator unit schematic - notice that the unit is simplified for better readability

Output kernel

This kernel reads from the root of the comparator unit tree and writes sequentially to memory. Also it has a streaming memory interface.

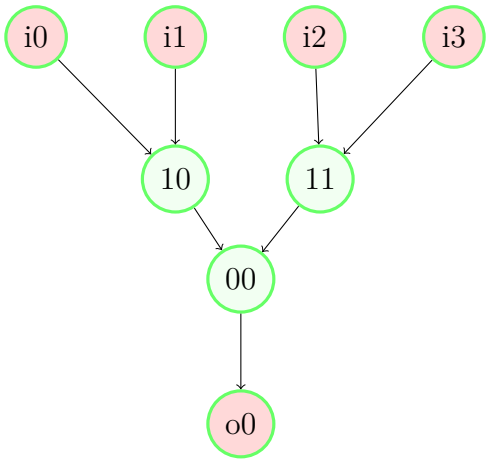


Figure 4.22: Mergesort design $N = 4$

Sorting behavior

A limitation of this sort is that it only can compare and sort 64 elements simultaneously. To sort more elements than that, it requires to receive the elements sorted, since he compare the head of the list at each iteration . To sort large arrays, our kernel does an incremental sorting. Assuming an unsorted array, we send list of 1 element to each input node, producing sorted list of 64 elements. now we send back this list to produce list of 64^2 elements. We repeat until it is all sorted. Sorting more elements requires more iterations but the iteration number increase very slowly since the elements that can be merged grows exponentially. Sorting a non power of 64 requires to add padding. One way is to fit it to a power on the CPU allocating more memory and the other way is to add padding on the FPGA, adding code to the input kernels to generate the needed padding. We start implementing the padding on the FPGA but we run out of time so we keep it for future work.

Results

Since our implementation only operates with powers of 64 (including padding) we tested the powers that are less than 4Gb of memory as we can't allocate more. We wrote a CPU sort using MKL to compare with out algorithm. As can be seen in the figure 4.23 the tested values are relatively small except the last one (16777216). The FPGA is slower to MKL in all cases. The CPU have a speedup of 62.5% over the FPGA in the last case. We can see in the figure 4.24 that our approach was correct since we have an speedup of 8x over the original implementation.

In the other hand, FPGA energy consumption scales better. Is worst with very small values but it has a 2x speedup over the CPU in the last case.

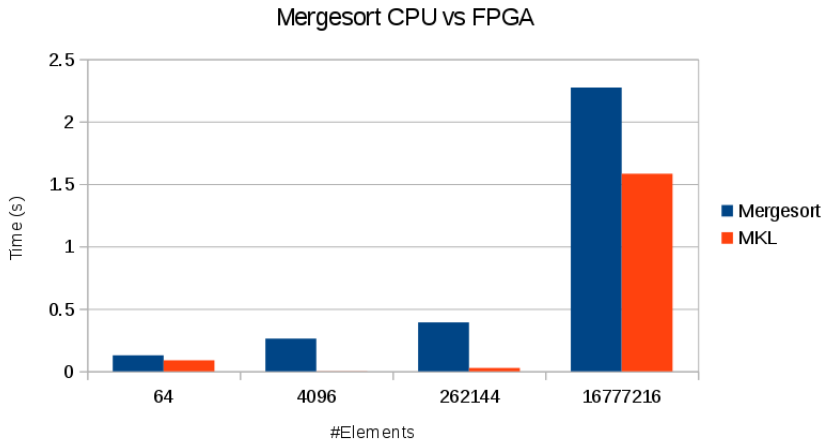


Figure 4.23: Time comparison between MKL and FPGA mergesort

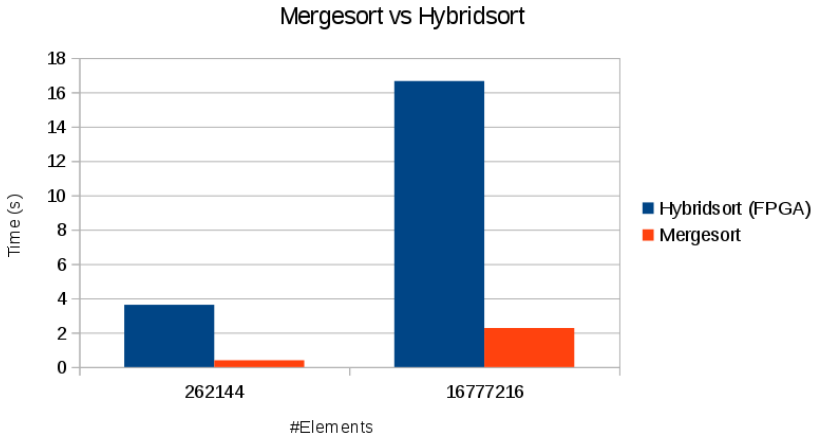


Figure 4.24: Time comparison between hybridsort on FPGA and mergesort on FPGA

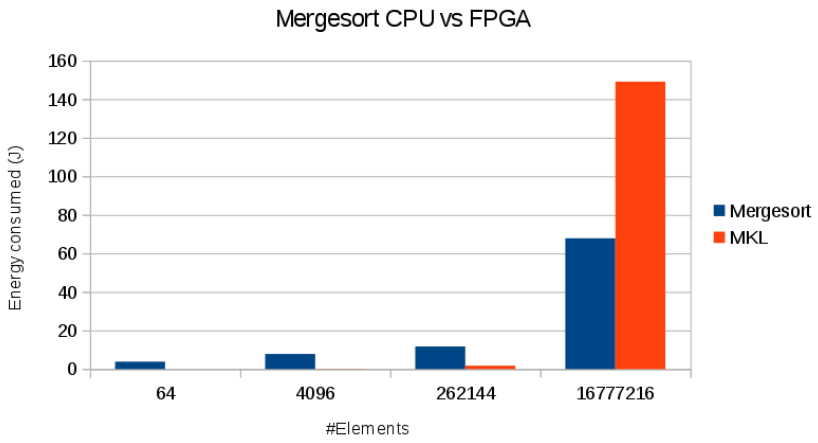


Figure 4.25: Energy comparison between MKL and FPGA mergesort

4.5 Hotspot2D

Algorithm description

Hotspot is a thermal model. It is useful to estimate the die thermal dissipation. It divides an area and compute the heat equation. As the algorithm is iterative, we compute a timestep each time. The computed areas are caculated in parallel.

$$T_{11} = T_{11} + step * (P_{11} + A + B + C + D + E) \tag{4.7}$$

$$A = \frac{T_{12} - T_{11}}{R_x} \tag{4.8}$$

$$B = \frac{T_{10} - T_{11}}{R_x} \tag{4.9}$$

$$C = \frac{T_{01} - T_{11}}{R_y} \tag{4.10}$$

$$D = \frac{T_{21} - T_{11}}{R_y} \tag{4.11}$$

$$E = \frac{T_{top} - T_{11}}{R_z} \tag{4.12}$$

Figure 4.26: Hotspot heat dissipation system

In the figure 4.26 we can see the equations solved, where T is temperature,

P is power and R is thermal resistance. This formula is solved many times to simulate the timesteps, where $step$ determine the precision of this timesteps.

Analysis and optimizations

We take the Rodinia [1] OpenCL implementation of Hotspot as our base implementation. The kernel is designed as a 2D-NDRange with the size of the cell (CellSize*CellSize). Every work-item have assigned a position of the cell and for each iteration, gets the neighbors temperature and computes its own new temperature. To compare the results, we used the OpenMP version from Rodinia.

Rewriting

Even though the kernel compiles fine, the implementation is pretty inefficient. Values like power are constants across the execution and they are stored in local memory instead of private memory. Also, temporal values use local memory instead of private memory. FPGA have plenty of private memory so we rewrite this parts to avoid the use of BRAMs.

Saturating resources

Now we have 2 options: apply the SIMD or replicate the compute units. We tried to unroll the main loop but the iterations have dependencies between them. Memory accesses can't be optimized by the SIMD optimization because their are not aligned. If we apply SIMD it is probably that we won't have enough bandwidth to feed the resources so, as we have many work-groups, so we expect that replicating the compute units it performs better. We tried 3 versions of it. One full SIMD, another full replication and another with a mix between them.

As we can see in figure 4.27, replicate the compute units was the correct approach, being nearly 2x faster than the SIMD approach on 8K grids and as fast as SIMD on the smallest grid. The mixed version is as fast as the compute units one . We can also see in figure 4.28 that the SIMD approach is the more power efficient for small grids, consuming less energy than the CPU. We can also see that, being slower than the CPU, we are always consuming less energy except for a big grid where the compute units version consumes nearly the same.

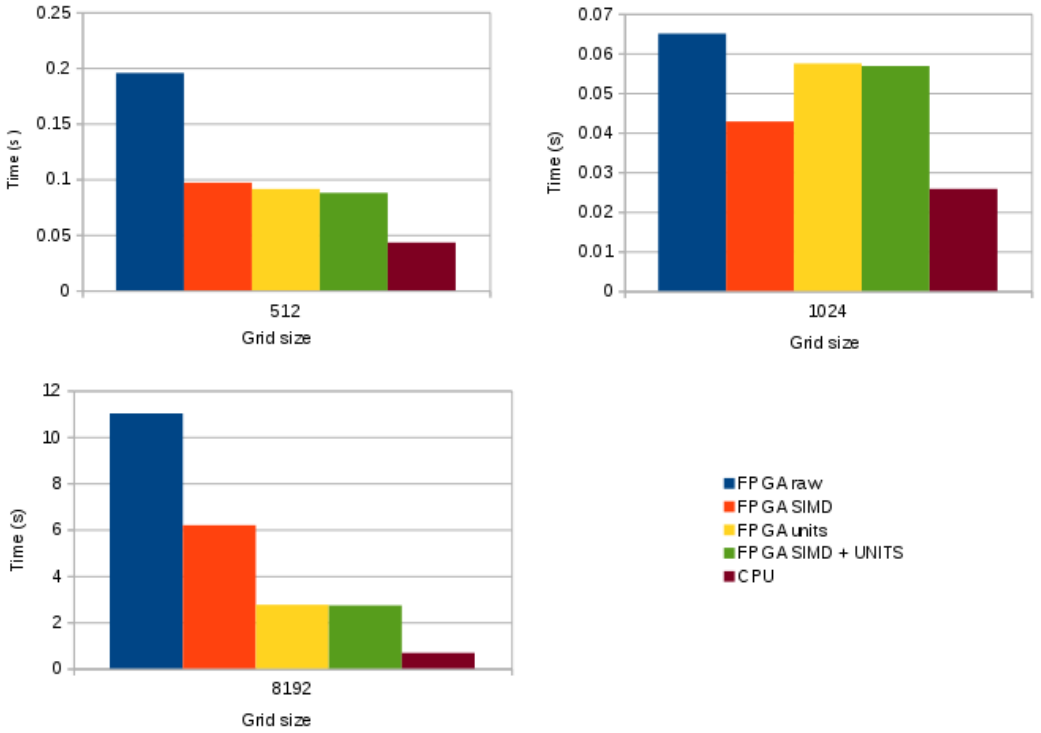


Figure 4.27: Time comparison between different hotspot FPGA optimizations and the CPU

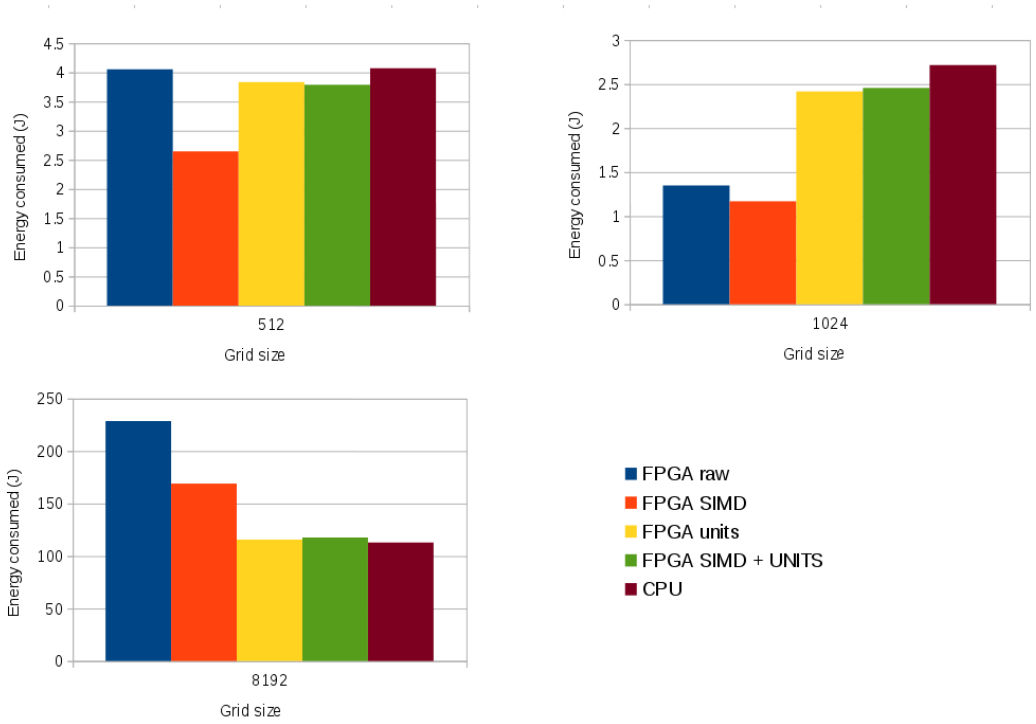


Figure 4.28: Energy comparison between different hotspot FPGA optimizations and the CPU

We tried the optimizations that the compiler can provide and we didn't get anything meaningful. What we can try is to tweak the algorithm a little bit. Since the work-groups work with cells, we can try to increase that cell size so a single work-group does more work. We were to pass from 16x16 cells to 64x64 and we saturate the rest of the FPGA with compute units. We test this optimization on the CPU too.

As we can see in figure 4.29, FPGA performance improves a little bit, specially for small benchmarks. That doesn't look much, but if we look at figure 4.30, we can see that now the FPGA consume half of the energy than the CPU or the best FPGA previous benchmark. The CPU, with this optimization, is even faster and consumes less, but not less than the FPGA with bigger cells.

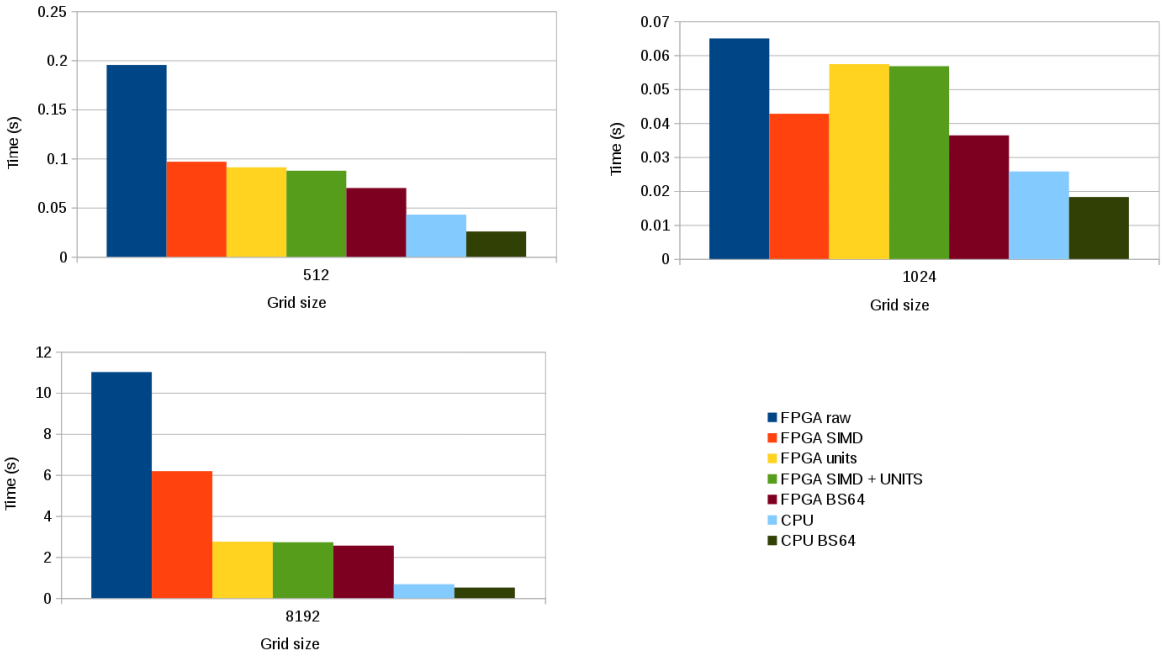


Figure 4.29: Time comparison between different hotspot FPGA optimizations and the CPU. This figure is like figure 4.27 but it don't have FPGA BS64 and CPU BS64 which use a block size of 64x64 instead of 16x16 elements

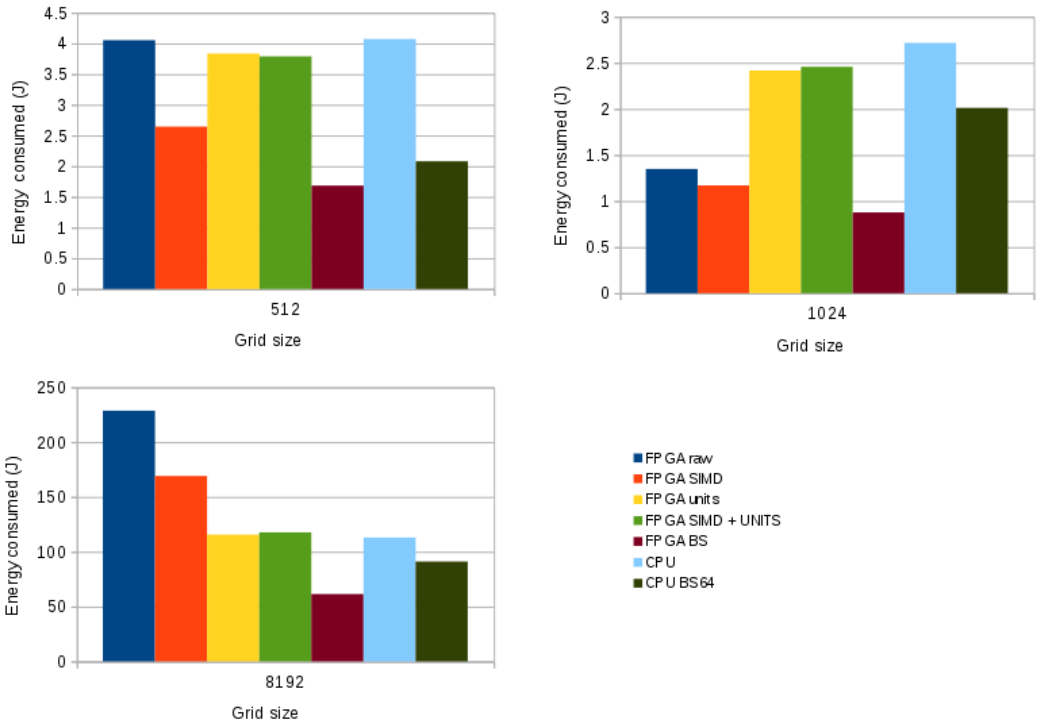


Figure 4.30: Energy comparison between different hotspot FPGA optimizations and the CPU. This figure is like figure 4.28 but it don't have FPGA BS64 and CPU BS64 which use a block size of 64x64 instead of 16x16 elements

4.6 Back Propagation

Algorithm description

Back Propagation is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. The application contains 2 phases: Forward phase, where the activations are forward propagated, and Backwards phase, where the error between the observed and requested value in the output layer is propagated backwards to adjust the weights.

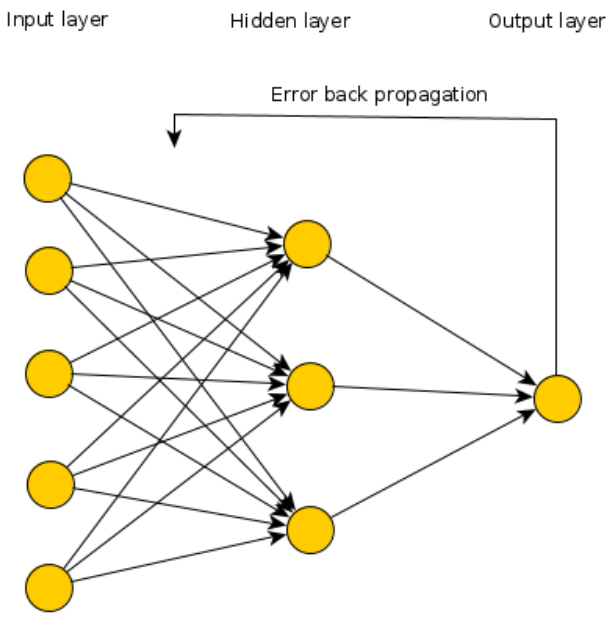


Figure 4.31: Back propagation algorithm

Analysis and optimizations

We start from the Rodinian [1] Back propagation OpenCL implementation. It is composed by 2 ND-Range kernels of 2 dimensions. Kernels requires some changes to compile on the FPGA since it contains variable size arrays. Arrays(BRAMS) size need to be known at compile time. With this porting done, the optimizations that we can do are not very extensive. All the kernels contains no loops (or fully unrolled ones) which is nice for an NDRange approach. We can go for 2 approaches here: Optimize with SIMD or optimize replicating kernels. To verify the difference between an NDrange and a single-thread kernel we also wrote a single-thread version. We will compare them with the Rodinian OpenMP version.

As we can see in the figure 4.32, the SIMD optimization is the fastest version, beating all the others including the CPU. The extra compute units optimization is slower than the raw one which doesn't make any sense. We think that this loss in performance can come from a bad work-group manager by the OpenCL runtime or because having more compute units, generate too many memory petition saturating the memory bus. We can see that the single-thread version (task) is the slowest one.

Looking at the figure 4.33, we can see that the single-thread version being the slowest kernel, is the kernel that have consumed less energy, followed by the raw kernel. The kernel with extra compute units it is the one that have consumed more energy.

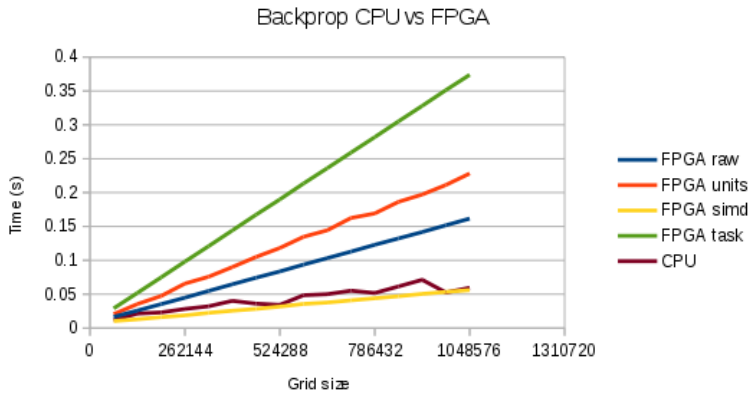


Figure 4.32: Backprop performance without optimizations, with SIMD, with kernel replication agains the CPU

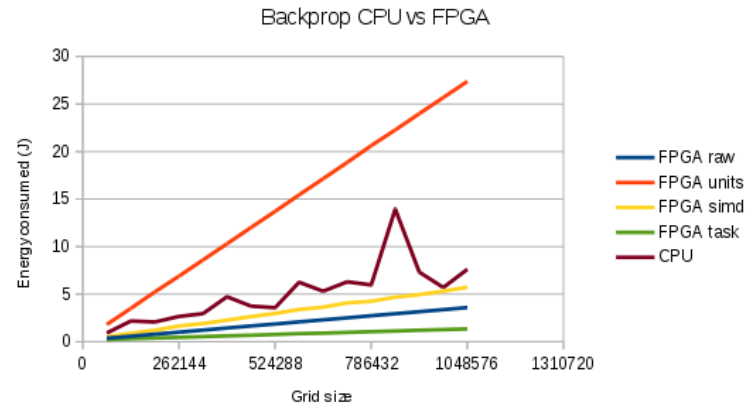


Figure 4.33: Backprop performance without optimizations, with SIMD, with kernel replication agains the CPU

4.7 AES-256

Algorithm description

AES is a standard encryption algorithm designed to be implementable in hardware. The algorithm splits the input data in blocks of 128 bits and applies the encryption algorithm one by one. Each block is the representation of a $4x4$ matrix.

The first step of the algorithm is to expand the key. The key is expanded to a 14 blocks of 16 bytes. This is because we need a block for each algorithm iteration. The iterations are determinate by the key size (256 bits requires 14 iterations). With the key expanded, the algorithm performs the required iterations of this sequence of operations.

1. Initial round
   (a) AddRound: xor between each byte and the expanded key
2. Rounds
   (a) SubBytes: substitute bytes based in a lookup table
   (b) ShiftRows: shift the rows
   (c) MixColumns: swap columns
   (d) AddRound
3. Last round
   (a) SubBytes
   (b) ShiftRows
   (c) AddRound

This is not enough to encrypt the data, since some sort of data (like images) leave traces of the raw data. To solve this, block cipher mode of operation where designed. Our implementation use the ECB, which is the raw AES algorithm. We decide to use this mode for simplicity. Porting this mode to a effective mode (CTR mode for parallel encryption/decryption) should not involve any performance impact and a minimal extra logic usage.

$$\begin{bmatrix} b_1 & b_2 & b_3 & b_4 \\ b_5 & b_6 & b_7 & b_8 \\ b_9 & b_{10} & b_{11} & b_{12} \\ b_{13} & b_{14} & b_{15} & b_{16} \end{bmatrix} \tag{4.13}$$

Analysis and optimizations

All the AES operations are simple and are easily implementable on hardware. There isn't much parallelism in block operations, but we can process blocks in parallel. Since the key expansion is done on the FPGA and it is shared by all the threads we only design a single thread version of AES instead of going for a NDRange. Without optimization, the AES doesn't fit on the FPGA because it has a lot of overhead for the loops. When we unrolled all the loops it fits perfectly and leaves us a little space for unrolling partially the external loop. To compare with it, we written a CPU version using OmpSs and Intel AES intrinsics.

We can see in the figure 4.34 that the FPGA is slower than the CPU which makes sense since the CPU has hardware support for AES, but if we look at the figure 4.35 we can see that even being like 10 times slower the FPGA is consuming less energy than the CPU.

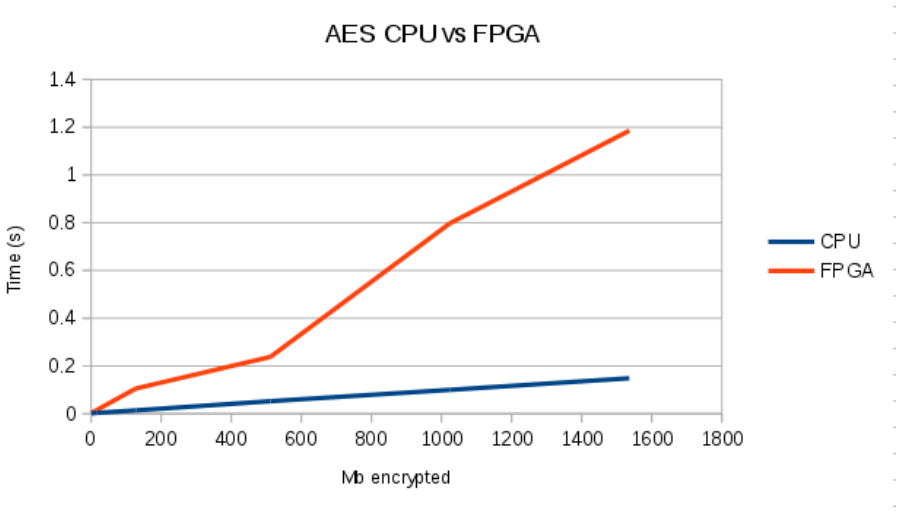


Figure 4.34: AES performance vs CPU

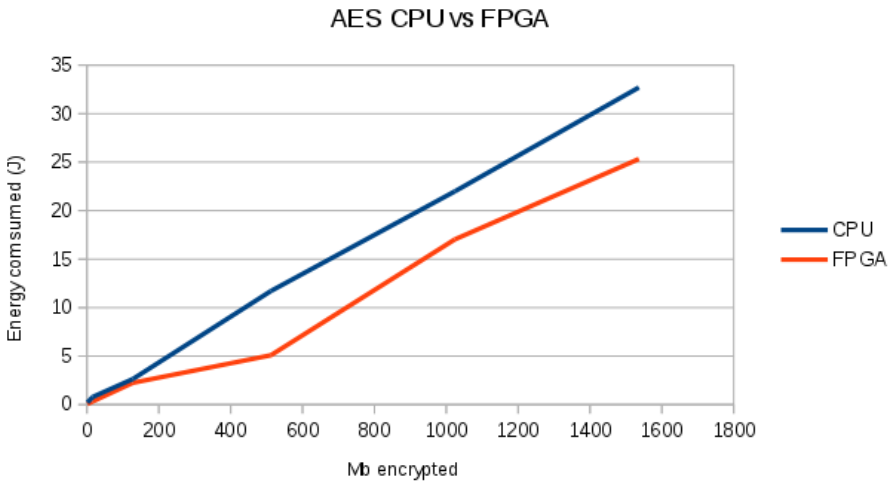


Figure 4.35: AES energy consumed vs CPU

Chapter 5

Results

5.1 Performance and energy results

We have presented a set of optimized benchmarks for the FPGA. In figure 5.1, we can see the speedup of the best FPGA version again the CPU for each benchmark. We can see that the only benchmarks that gets performance speedup are Nbody and Backprop.

Cholesky, Mergesort, Hotspot and AES couldn't beat the CPU. The CPU version of AES use AES CPU intrinsics so it is very hard to beat. Mergesort is really close to the CPU and maybe with bigger data input it can beat the CPU. CFD has indirect memory pointers and we learned that this ones performs very bad on an FPGA. Cholesky is really slow in this plot because the FPGA is just slightly faster doing GEMMs than a single CPU core. Since the CPU can do 24 GEMM in parallel the CPU es much faster than the FPGA.

In the other hand, 4 of 6 kernels has speedup on energy consumption. The most notable are NBody and backprop which also have performance speedup. Other kernels like mergesort, hotspot and AES are more power efficient than the CPU but they don't have performance speedup.

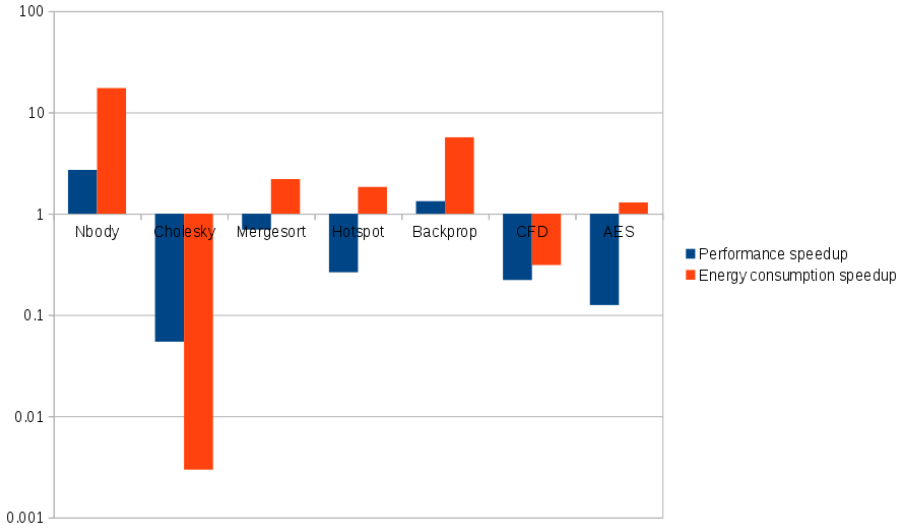


Figure 5.1: Normalized speedup in performance and energy consumption from FPGA vs the CPU for all the benchmarks

5.2 Learned lessons

Over this work we have learned a lot on how to do a porting to FPGA and which are the programs that are ideal for the FPGA.

When you port a kernel to a FPGA:

1. Look if the part that you want to port it potentially can work well on the FPGA. A code with long latency operations (like nbody or backprop), simple loops and sequential memory access should perform fine.

2. When you start porting a kernel from C/C++, it is easier to start from a task. If you see that you get sub-loops or you get complicate memory access probably you will get better results moving to a NDRange. In Backprop, we moved from NDRange (raw) to a task to check the difference and we get a 3x drop in performance. That's is because we are adding a loop that has more overhead than a NDRange.

3. Try to do memory access wider and sequential. Access memory sequentially allow to generate simple memory interfaces or to exploit the cache if there is one. For example NBody have 99% hit rate on its caches.

4. Access global memory with index with the form $i + k$ (constant increment and a constant). Using this kind of index generates streaming or

semitraming intefaces. CFD access the memory by columns and also do indirect access, that leads to losing nearly all the time waiting for the memory to respond (75%), so this is critical to get good performance.

5. Eliminate any loop that you can unrolling it or using NDRange and partially unroll the rest. Specially do it on NDRange kernels were loops are not pipelined. For example, NBody is an NDRange with a single loop which does all the work. Since the loops are not pipelined the optimal case is that we unroll it completely so the loop is removed and the iterations can be pipelined because there is no loop anymore. As the loop is not bounded we can't removed so we unrolled partially providing more performance because we are doing more parallel work and doing more memory access.

6. Avoid indirect memory access. We seen on CFD that the FPGA have performance issues with it.

7. Use private memory over local or global memory. Avoid global memory as much as you can. Global memory generate costly and slow interfaces, and they are generated for each memory access that is written. Local memory generate BRAMs which are efficient but we don't have many. Private memory are registers which we have plenty of them and they are quite fast.

8. If your kernel is too complex to be optimized, think if you can split it in multiple kernels that communicates by pipes. Mergesort is a good example of it, splitting it on multiple kernels let us generate streaming interfaces and split the work in very simple units creating a more tailored pipeline and achieving an 8x over the initial implementation.

Chapter 6

Conclusions

We have presented a set of benchmarks that have been ported to a shared memory architecture with an Intel Xeon and an Intel Arria 10 using OpenCL. We have detailed which decisions were made to optimize every OpenCL kernel and we compared the results with a parallel CPU version.

We have evaluated the ported kernels, showing performance and energy consumption results. We can conclude that the FPGA provides great power efficiency. Hotspot and Mergesort, kernels that are slower than the CPU, consumes less than it.

In some cases, the FPGA can provide performance as in NBody and Backprop. Nbody and Backprop are really simple and they have very long latency operations that the FPGA can optimize very well since it is creating a custom pipeline for the problem.

We learned how a program needs to be ported to provide good performance as we described in the Learned Lessons section.

We can also conclude that OpenCL for FPGA provides a fast way to port programs to an FPGA, providing some flexibility to implement low level designs like we did with the Mergesort, but abstracting from the complexity of a HDL.

Acknowledgements

I would like to thanks to my professors Xavier, Carlos and Daniel for giving me the opportunity to base my master thesis on FPGAs. I also have to thanks their wise guidance and their patience with me.

Thanks to my friends: Quim, Joan and Eudald that always were there when I needed them.

I also have to thanks my workmates: Dani, Artem, Ivan, Aleix, Jordi, Miquel and Marc for encouraging me in my bad moments.

And finally, a special thanks to Alice for her extreme patience with my work and with me.

Bibliography

[1] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009*, pages 44–54, 2009. ISBN 9781424451562. doi: 10.1109/IISWC.2009.5306797.

[2] Doris Chen and Deshanand Singh. Fractal Video Compression in OpenCL: An Evaluation of CPUs, GPUs, and FPGAs as Acceleration Platforms. URL `https://www.altera.com/content/dam/altera-www/global/zh{\_}CN/pdfs/products/software/opencl/ieee-cp-fractal-video-compression.pdf`.

[3] ALEJANDRO DURAN, EDUARD AYGUADÉ, ROSA M. BADIA, JESÚS LABARTA, LUIS MARTINELL, XAVIER MARTORELL, and JUDIT PLANAS. OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES. *Parallel Processing Letters*, 21(02):173–193, 2011. ISSN 0129-6264. doi: 10.1142/S0129626411000151. URL `http://www.worldscientific.com/doi/abs/10.1142/S0129626411000151`.

[4] Zheming Jin, Kazutomo Yoshii, Hal Finkel, and Franck Cappello. Evaluation of CHO Benchmarks on the Arria 10 FPGA using Intel FPGA SDK for OpenCL. Technical report, Argonne National Laboratory (ANL), Argonne, IL (United States), may 2017. URL `http://www.osti.gov/servlets/purl/1372106/`.

[5] Zheming Jin, Hal Finkel, Kazutomo Yoshii, and Franck Cappello. Evaluation of a Floating-Point Intensive Kernel on FPGA. pages 664–675. Springer, Cham, aug 2018. URL `http://link.springer.com/10.1007/978-3-319-75178-8{\_}53`.

[6] Yingyi Luo, Xianshan Wen, Kazutomo Yoshii, Seda Ogrenci-Memik, Gokhan Memik, Hal Finkel, and Franck Cappello. Evaluating irregu-

lar memory access on OpenCL FPGA platforms: A case study with XSBench. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, sep 2017. ISBN 978-9-0903-0428-1. doi: 10.23919/FPL.2017.8056827. URL http://ieeexplore.ieee.org/document/8056827/.

[7] Fahad Bin Muslim, Liang Ma, Mehdi Roozmeh, and Luciano Lavagno. Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. *IEEE Access*, 5:2747–2762, 2017. ISSN 2169-3536. doi: 10.1109/ACCESS.2017.2671881. URL http://ieeexplore.ieee.org/document/7859319/.

[8] Kohei Nagasu, Kentaro Sano, Fumiya Kono, and Naohito Nakasato. FPGA-based tsunami simulation: Performance comparison with GPUs, and roofline model for scalability analysis. *Journal of Parallel and Distributed Computing*, 106:153–169, aug 2017. ISSN 0743-7315. doi: 10.1016/J.JPDC.2016.12.015. URL https://www.sciencedirect.com/science/article/pii/S0743731516301915.

[9] Khronos Opencl. OpenCL Specification. *ReVision*, pages 1–385, 2009. ISSN 13596454. doi: 10.1016/j.actamat.2006.08.044. URL http://scholar.google.com/scholar?hl=en{\&}btnG=Search{\&}q=intitle:OpenCL+Specification{\#}0.

[10] Amulya Vishwanath. Enabling High-Performance Floating-Point Designs. *Intel Whitepaper*, 2016. URL https://www.altera.com/content/dam/altera-www/global/en{_}US/pdfs/literature/wp/wp-01267-fpgas-enable-high-performance-floating-point.pdf.

[11] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. Energy Efficient Scientific Computing on FPGAs using OpenCL. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pages 247–256, New York, New York, USA, 2017. ACM Press. ISBN 9781450343541. doi: 10.1145/3020078.3021730. URL http://dl.acm.org/citation.cfm?doid=3020078.3021730.